Regular Expressions

Introduction

The terms *regular expression*, *regex* and *regexp* are 100% synonymous. I call them *regexes*.

> Gotcha: The character is both an escape and an unescape character depending on the context.

Tools

1. https://regex101.com/
2. https://regexr.com/
3. Any JavaScript REPL (e.g. repl.it, jsfiddle, codepen, F12...)
4. Many more

Introduction

Regex has three main uses:

1. String validation.
2. Extracting sub-strings.
3. Finding and replacing sub-strings.

Unfortunately there are many variations of regex. .NET, JavaScript, Perl and many more. Make sure you're using the one you think you are. I'm going to use JavaScript and use RegExr for evaluating and examining them.

In JavaScript regexes are so common that we have a literal form as well as the, largely unused, object notation. We actually have three choices when creating a regex.

JavaScript Syntax

Whether JavaScript or another flavour all regexes have two parts:

1. A pattern.
2. A list of flags.

The pattern defines most of the logic but the flags can allow us to simplify our expression. The following table is a description of all the valid flags in the order (that I think) they are most used

Flag	Description
i	Ignore case.
s	Allow the `.` to match new lines.
m	Make this a multiline search. This causes `^` and `$` to match the start and end of lines.
g	Global.
d	Generate indices for match groups.
u	Unicode
y	Sticky algorithm.

> Personally, I don't think you need to know `g`, `d`, `u` or `y` unless you're planning on writing mental regexes. If you are planning on writing a mental regex, don't because nobody but you will be clever enough to understand it (including the future you).

Syntax

JavaScript lets us declare a regex in several ways. There's no reason to use anything other that the literal form though, unless you're dynamically generating the pattern or the flags.

Literal

```
/my-regex-pattern/flags
```

Object

```
new RegExp('my-regex-pattern'[, 'flags'])
new RegExp(/my-regex-pattern/[, 'flags'])
    RegExp('my-regex-pattern'[, 'flags'])
    RegExp(/my-regex-pattern/[, 'flags'])
new RegExp(/my-regex-pattern/flags)
    RegExp(/my-regex-pattern/flags)
```

> WRT the object form, you can add flags to both the regex literal and the the second argument but only the flags passed, however the flags passed in the literal will be ignored.

Executing

At a basic level, validation work be reading from each character in the candidate string, determining if it matches the pattern. If it does then the character is appended to a cache of matching character strings and the engine will decide whether of not to continue.

The simplest regex is just a test for contains.

```
/abc/
```

Several functions can be invoked to use this regex, some of them are members of the string, some members of the regex and some the `Regex` object. The functionality of these functions overlaps quite alot and there's quite a bit of duplication. We'll just look at two off the regex object for now.

```
const regex = /abc/;
const candidateMatch = 'abc';
const candidateUnmatched = 'x';

const testResultMatched = regex.test(candidateMatch);
const testResultUnmatched = regex.test(candidateUnmatched);

const execResultMatched = regex.exec(candidateMatch);
const execResultUnmatched = regex.exec(candidateUnmatched);

console.group('test');
console.dir(testResultMatched);
console.dir(testResultUnmatched);
console.groupEnd();

console.group('exec')
console.dir(execResultMatched);
console.dir(execResultUnmatched);
console.groupEnd();
```

This will output:

```
test
  true
  false
exec
  [ 'abc', index: 0, input: 'abc', groups: undefined ]
  null
```

So `.test()` gives us a simple boolean whereas the `.exec()` gives us an array of objects containing metadata about the match (or `null` is there is no match).

`/abc/` will match any string that contains a substring of `'abc'`, so the following also match:

```
const regex = /abc/;
const tests = [
    {input: 'abc'},
    {input: ' abc'},
    {input: 'abcc'},
    {input: 'aabcc'}];

for (let i = 0; i < tests.length; i++)
    tests[i].result = regex.test(tests[i].input);

console.table(tests);
```

Produces:

(index)	input	result
0	`'abc'`	`true`
1	`' abc'`	`true`
2	`'abcc'`	`true`
3	`'aabcc '`	`true`

String Validation

To validate a string we can create much more complicated expressions.

Repeating characters

We have a few symbols we can use to tell the regex engine that we want to look for repeating characters.

Metacharacter(s)	Description
`*`	Match zero or more instances.
`?`	Match zero or one instance.
`{[[min],[max]]}`	Between `min` and `max` (*inclusive*). Either is optional.

If we knew exactly how many repeating characters then we could just type them, this is fine when there aren't many characters but very difficult to read when there's more than a few. We might also want to allow a a number of characters between two limits which we couldn't do by typing.

> Technically we could use an `OR` to have a variable number of characters but there's no real world scenario where we'd do that.

A common regex is matching a URL. The two common schemes are `http://` and `https://` so we want to match `http` and optionally one or more `s` characters, followed by the `://`. We can use the `?` metacharacter to achieve this.

> Remember that the `/` character delimits the regex datatype and it's flags. The escape character in regexes is .

```
/^https?:\/\/$/
```

(index)	input	result
0	`http://`	`true`
1	`https://`	`true`

(index)	input	result
2	`htt://`	`false`
3	`http s://`	`false`

Metacharacters

The regex as we have it will match any string *containing* the pattern. This means that the regex as we have it still matches some invalid strings.

(index)	input	result
0	`foohttp://bar`	`true`
1	`https://`	`true`
2	`ftp://http://`	`true`

Metacharacters are place holders for non-printable positions such as the start of a string, the end of a string, word boundries and others. To fix the regex above we need to add the metacharacters for the start and end of the string.

```
/^https?://$/
```

(index)	input	result
0	`'http://'`	`true`
1	`'https://  '`	`false`
2	`'foohttps://  '`	`false`
3	`'ftp://https://  '`	`false`

Matching the domain brings some new challenges. Let's just work with simple examples first, addresses with just a TLD and a SLD.

- `http://example.com`
- `https://example.com`
- `http://example.es`
- `https://example.es`

- `foo-bar.com`
- ...

We need to match any valid character for a SLD and then for a TLD, we also need to escape the `.`.

We'll *pretend* that valid URLs are all the alphanumeric characters.

To achieve this we need to introduce a new concept, *Character Classes*.

Character Classes

A character class is a set of characters. We use them when we want to say that any character in a set is valid. We declare a character class by wrapping the characters in square brackets. For example, `[abc]` would match the characters `a`, `b` or `c`. We can apply the repetition modifiers above as if this were a single character.

For our URL example, to say that we want any alphanumeric character we could write:

```
/abcdefghijklmnopqrstuvwzyzABCDEFGHIJKLMNOPQRSTUVWZYZ0123456789]*/
```

This is a lot to type out, so we have some shortcuts. The first option is to use ranges.

```
/[a-zA-z0-9]/
```

This is much easier to read, but we can to better. Remember the flags? We can add the `i` flag to make it case insensitive so that we don't need to declare both cases.

```
/[a-z0-9]/i
```

We can make yet another improvement, `\d` is a shorthand for any number character (equivilent to `[0-9]`).

```
/[a-z\d]/i
```

Finally, there one more improvement we can make. The `\w` character class matches any alphanumeric character of any case.

```
/\w/
```

```
/^https?:/\w*\.[a-z]{2,3}$/
```

OR with words

The character classes are akin to a long boolean `OR` over a set of single characters. We can also perform an `OR` on strings. The URL above could have been written as an `http` or `https`. The way we have written it is more succinct than writing an entire `OR` but there are other cases where might not be able to just have an optional character. If we wanted to confirm that a filename has an image extension then we would have to check that it ends with a `.jpg`, `.gif`, `.bmp` etc. To achieve this we use the syntax:

```
/(\.gif|\.png|\.bmp)$/
```

(index)	input	result
`0`	`file.jpg`	`true`
`1`	`file.gif`	`true`
`2`	`file.bmp`	`true`
`3`	`my.file.jpg.bm`	`true`
`4`	`not-an-image.jpg.txt`	`false`

Capturing groups

There's times when we need to look for repetitions that aren't next to each other, for example we might want to check that a list of filenames all have the same extension.

Let take a simple example. We'll have some arbitrary string that's repeated three times and separated by a `,` and then a `-`.

e.g.:

- `a,a-a`
- `foo,foo-foo`

In the previous section we used `(` and `)` to delimit the options in the `OR` . This actually create a capturing group and we can refer back to capturing groups in other parts of the regex. The simplest syntax for referencing a capturing group is a followed by an integer representing the ordinal position of the group to retrieve.

in the example above we create a capturing group around the text prior to the `,` and reuse it with `\1` .

```
/^(\w+),\1-\1$/
```

(index)	input	result
0	'a,a-a'	true
1	'foo,foo-foo'	true
2	'foo,bar-foo'	false

With many capturing groups we just increment the index.

```
/^(\w+),(\w+) Reversed is: \2,\1$/
```

(index)	input	result
0	'foo,bar Reversed is: bar,foo'	true
1	'1,2 Reversed is: 2,1'	true
2	'foo,bar Reversed is: foo,bar'	false

There's an obvious problem here. It's *horrible* to look at with any more than a handful of references. Altering the regex in the future is impossible because the groups are referenced ordinally. I nicer solution is using *named capture groups*.

```
/^(?<first>\w+),(?<second>\w+) Reversed is: \k<second>,\k<first>$/
```

> Node has problems with the `\k<...>` syntax.