Date, Regular Expressions

October 2022

Agenda

1 LETTER TO READER

2 INTRO

3 DATE

4 REGULAR EXPRESSIONS

You will need 3 important technical skills in projects (and on interviews):

Theoretical knowledge

Working experience

Hands-on skills

Let me go through this, one by one, because it could be important for you.

Friends, Romans, countrymen, lend me your ears

Theoretical knowledge

This is it. This lecture series tries to provide you what you need to know at first.

You also need to know (you may already realised), that this field stretches infinitely.

While it is very useful to get familiar with the *cyclomatic complexity* or with the *big O notation*, you'd need to be focused.

Don't get me wrong: you'd need to understand the basics – the difference between the *position: relative and absolute* is essential.

Ok, moving on …

Working experience

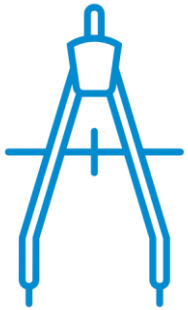
This will be your main asset. However, you'll need years to build it. Not one and not two.

Seriously. And still then, you probably won't succeed. Why? There are many reasons:

Working on wrong projects, with wrong practices, with wrong toolsets (lack of unit tests, hello there), with wrong processes you will master only that: how to do it wrongly.

There are other reasons, but for now, you can be relieved: we don't expect working experience from you at this point - so we can

move on …



Hands-on skills, coding skills, algorithmic skills. You name it.

While there are subtle differences (you don't need to write bubble sort on projects, nor RSA algorithms), you can be absolutely sure, that this will be required.

Both on interviews and on projects as well.

What does it mean, exactly? You will face – usually small – problems, and you need to implement them, easily.

Like handling the wheel and clutch, without even thinking about it (there is no automatic transmission here, sorry for that).

For that reason, you have homework here. The bad news: it probably won't be enough. The good news: while you'd need to put a lot effort in it, it could be improved.

Nope, working on pet projects won't do that. It is fine, and could improve your other skills, but now we need to work on your hands-on skills as much efficiently, as it is possible to do.

I'll tell you how to do that, exactly…

You need to work on exercises. On hundreds of exercises.

There are many similar websites to codewars, still, I'd choose it.

How to do it?

Pick a topic, such as strings in JavaScript: Kata | Codewars.

I can see 58 kata there with 8 kyu (the easiest level), so your task is very simple (*we don't like complexity, do we?*): solve them all.

Then pick another topic (e.g., arrays) and do the same.

You can step up with the difficulty (as you progress, you will need to), but please do it only when you are able to solve the exercises so fast, that you will be bored to read the problem.

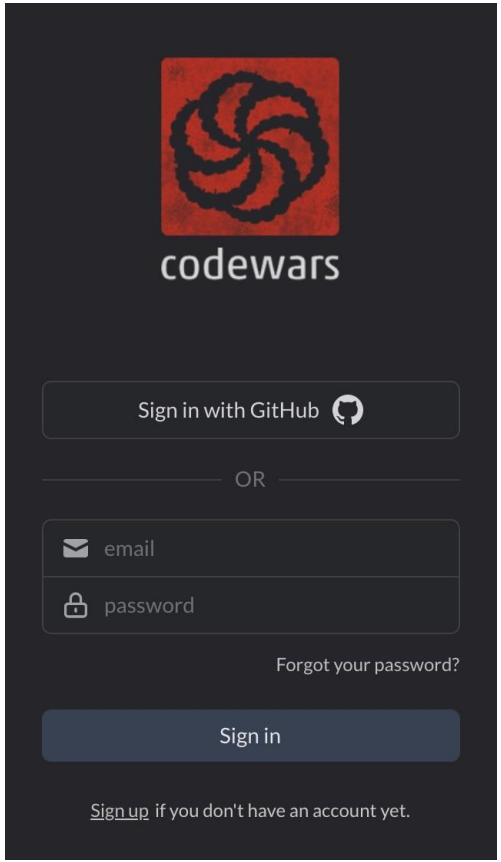
You don't need to solve complex problems there. You'll need to solve easy (= relatively easy to your level) katas, but literally hundreds of them.

Is this mandatory?

Nope. We do understand, that you have a family to care of, and generally, time is always a limited resource.

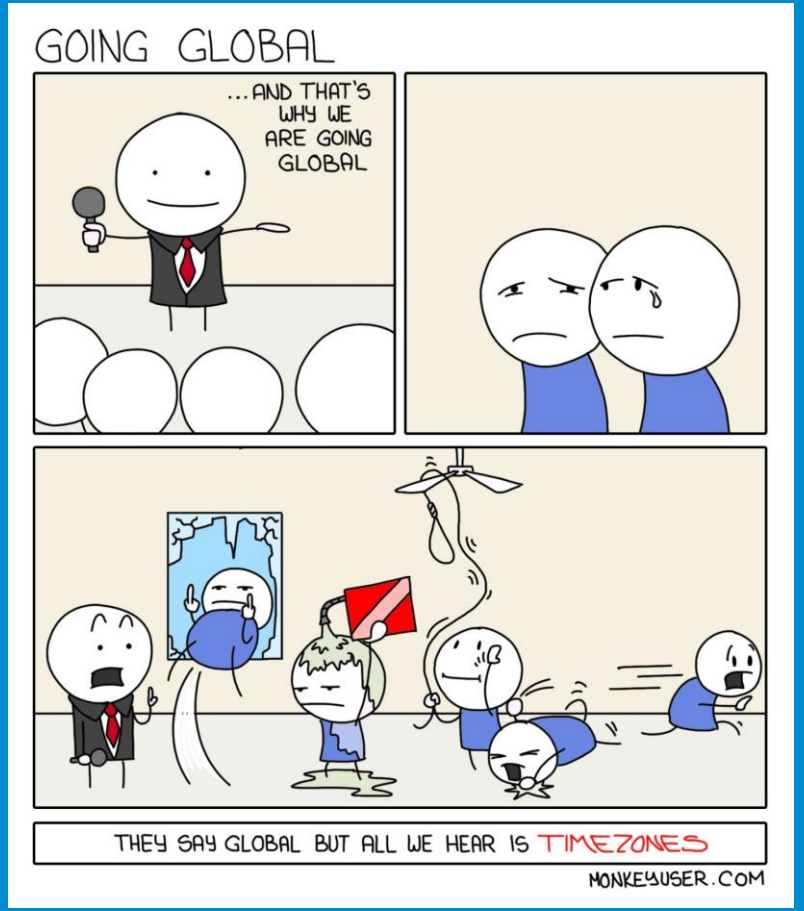
Still, if you'd ask what to do, this would be that. Ahh,

btw, the lectures are important, too ;)



register now. really.

Date


GOING GLOBAL
...AND THAT'S WHY WE ARE GOING GLOBAL
THEY SAY GLOBAL BUT ALL WE HEAR IS TIMEZONES
MONKEYUSER.COM

Epoch date

We cannot talk about dates in JavaScript without understanding the Unix Epoch

Essentially, handling the date in JavaScript means (somewhat similarly to Unix) counting the milliseconds from this time.

And that's all.

while the unix timestamp represents seconds, the JavaScript milliseconds

JavaScript date – well, not the best part

Well, if this is that simple, *what could go wrong*?

The problem is exactly like that: it is too simple. And while JavaScript provides several methods for handling dates, it lacks to address the problem properly.

It can be understood - it is a complex task, and even libraries failed in their attempts.

Still, it is something that has to do, and until then, your team could only ask: please don't take this topic lightheartedly (even when using libraries – or, especially then)

"JavaScript Date is broken in ways that cannot be fixed without breaking the web. As the story goes, it was included in the original 10-day JavaScript engine hack and based on java.util.Date, which itself was deprecated in 1997 due to being a terrible API and replaced with a better one.

The result has been for all of JavaScript's history, the built-in Date has remained very hard to work with directly."

JavaScript time = milliseconds from Epoch

Let's start with that innocent looking 3 characters: UTC. UTC means: Coordinated Universal Time.

> *UTC is the primary time standard by which the world regulates clocks and time. It is effectively a successor to Greenwich Mean Time (GMT).*

It also means, that the time we'd need to handle will be time zone dependent.

Time measurement in ECMAScript is analogous to time measurement in POSIX, in particular sharing definition in terms of the proleptic Gregorian calendar, an epoch of midnight at the beginning of 01 January, 1970 UTC, and an accounting of every day as comprising exactly 86,400 seconds (each of which is 1000 milliseconds long).

let's see an example!

The time zone problem

Let’s say, our client wants to have promotion about this nice poster, right?

They would like like to have this sale only for one day. Midnight to midnight.

The question is: what does that mean, exactly? What time zone it is?

Is it the time zone of the customer? Or of the headquarter of the client? Or the server? And then, which server, the app or the database?

The answer is: all of these. (At least from developer’s perspective). The chance, that your team would need to maintain the time and keep the transaction synchronized, is pretty high, so be prepared.

Date object

Date in JavaScript (surprise) is an object

Or more precisely, is a number (milliseconds from epoch), and an object around it. A bit of math:

A time value that is a multiple of 24 × 60 × 60 × 1000 = 86,400,000 represents a day.

A kind note from MDN:

It's important to keep in mind that while the time value at the heart of a Date object is UTC, the basic methods to fetch the date and time or its components all work in the local (i.e. host system) time zone and offset.

see? local time zone...

the Date constructor can be called without new, returning a *string* with the current day!*

why? because it does not care about parameters

```
> new Date(0);
Thu Jan 01 1970 01:00:00 GMT+0100 (Central European Standard Time)
> typeof new Date(0);
"object"
> Date(0);
"Sun Mar 12 1989 00:56:50 GMT+0100 (Central European Standard Time)"
> typeof Date(0);
"string"
```

*yep, today is the day, when the web was born

Date constructor

A new date can be created with calling a Date constructor, which consumes almost absolute everything

please appreciate the nuances: the month starts with zero, the day with 1, both can be negative

```
> new Date(-2042586000000);
Tue Apr 11 1905 00:00:00 GMT+0100 (Central European Summer Time)
> new Date("April 11, 1905");
Tue Apr 11 1905 00:00:00 GMT+0100 (Central European Summer Time)
> new Date(1905, 3, 11);
Tue Apr 11 1905 00:00:00 GMT+0100 (Central European Summer Time)
> new Date(new Date("April 11, 1905"));
Tue Apr 11 1905 00:00:00 GMT+0100 (Central European Summer Time)
```

Date methods

The date object does have zillion methods

for extracting date parts, for manipulating the date, for converting to string.

Basically, this is how they work:

1., you need a date to start with

```
> let lullaby = new Date(-2042586000000);
< undefined
```

4., create a new date

```
> new Date(lullaby.setFullYear(2021)).getFullYear();
< 2021
```

2., manipulate directly on the date object

3., return a timestamp

5., extract a part

6., finally: convert it to a string

```
> lullaby.toDateString
< f
```

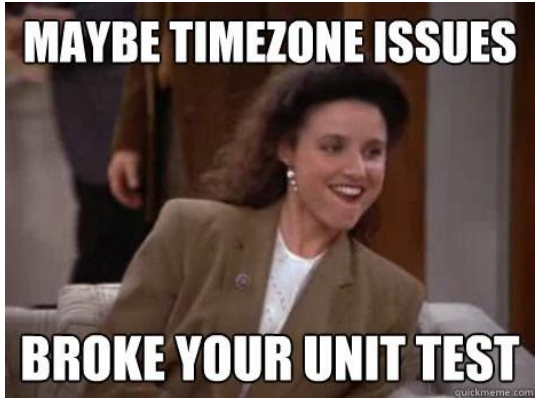
toDateString Date
toGMTString
toISOString
toJSON
toLocaleDateString
toLocaleString
toLocaleTimeString
toString
toTimeString
toUTCString
constructor
isPrototypeOf Object
__proto__

GMT, ISO, UTC, Locale…

Date methods

As we love the circular narratives, we finish where we started:

this way we can extract the primitive value behind the date object, the milliseconds elapsed from Epoch.

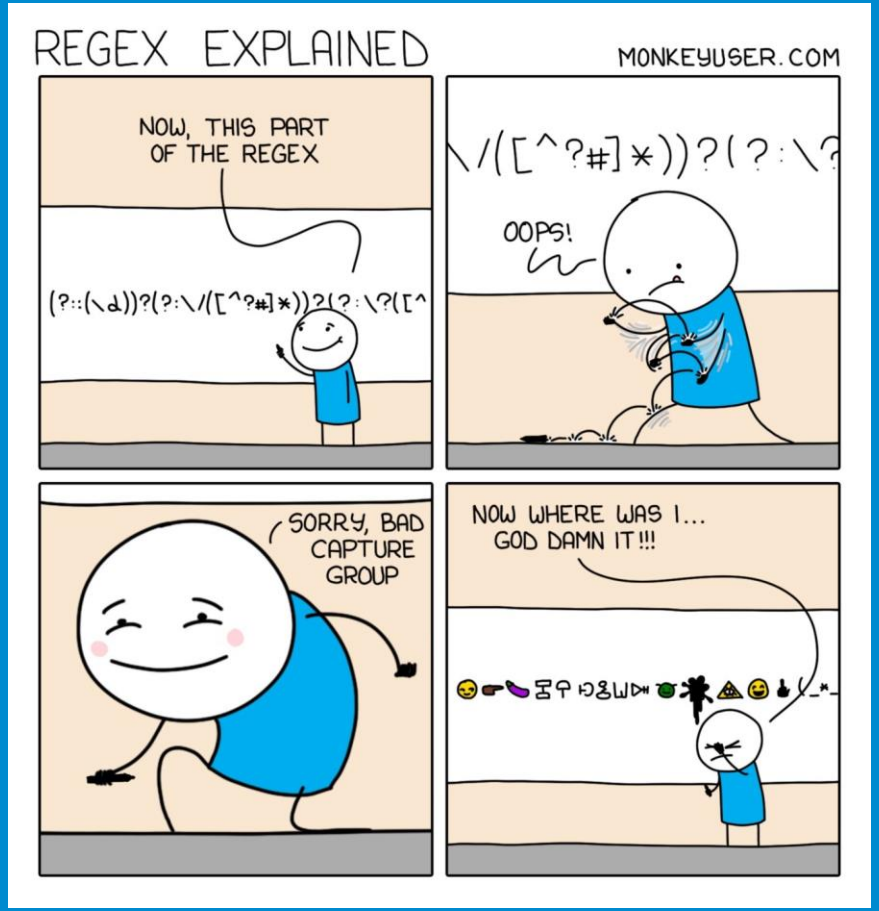


been there, done that…

these are equivalent according to the specification

```
> new Date(42).valueOf();
< 42
> new Date(42).getTime();
< 42
```

Regular Expressions



Regular Expressions (regex)

Regular expressions are patterns used to match character combinations in strings.

RegExp (pascal case) = the JavaScript constructor
regexp, regex (lower case) = regular expression, in general

Practically, a regex is used to

- extract specific parts from a string
- check, whether a string conforms to a pattern
- replace parts of a string

In JavaScript, regular expressions are also objects.

```
> /^Road[a-zA-Z?,'.\s]*Roads\.$/.test("Where We're Going, We Don't Need Roads.");
```

regular expressions are patterns

and at the end of this lecture, you should understand this regex completely

A regex declared with literal is still an object, too

object

```
> let regex = /scott/ig;
undefined
> typeof regex;
"object"
> regex.flags
```

it does have properties, some focused on flags

```
lastIndex
compile                RegExp
constructor
dotAll
exec
flags
global
ignoreCase
multiline
source
sticky
test
toString
unicode
hasOwnProperty         Object
isPrototypeOf
propertyIsEnumerable
toLocaleString
valueOf
__defineGetter__
```

.source provides the regex as a string

.exec and *.test* are important methods

```
> let regex = /scott/ig;
undefined
> regex.source
"scott"
> regex.flags
"gi"
> regex.lastIndex
0
> regex.test("Scott");
true
> regex.lastIndex
5
> regex.lastIndex = 0;
0
> regex.exec("Scott");
▶ ["Scott", index: 0, input: "Scott", groups: undefined]
> regex.exec("Scott");
null
```

a RegExp instance is stateful!

RegExp vs regex literal

Regular expressions can be created in 2 ways:

with RegExp constructor →

```
> let regex = new RegExp("^delorean", "i");
< undefined
```

with regexp literal →

```
> let regex = /^delorean/i;
< undefined
```

there is an important use case for the RexExp constructor…

RegExp vs regex literal

What to use then?

A RegExp constructor's parameter could be a string, therefore we can assemble the regex programmatically.

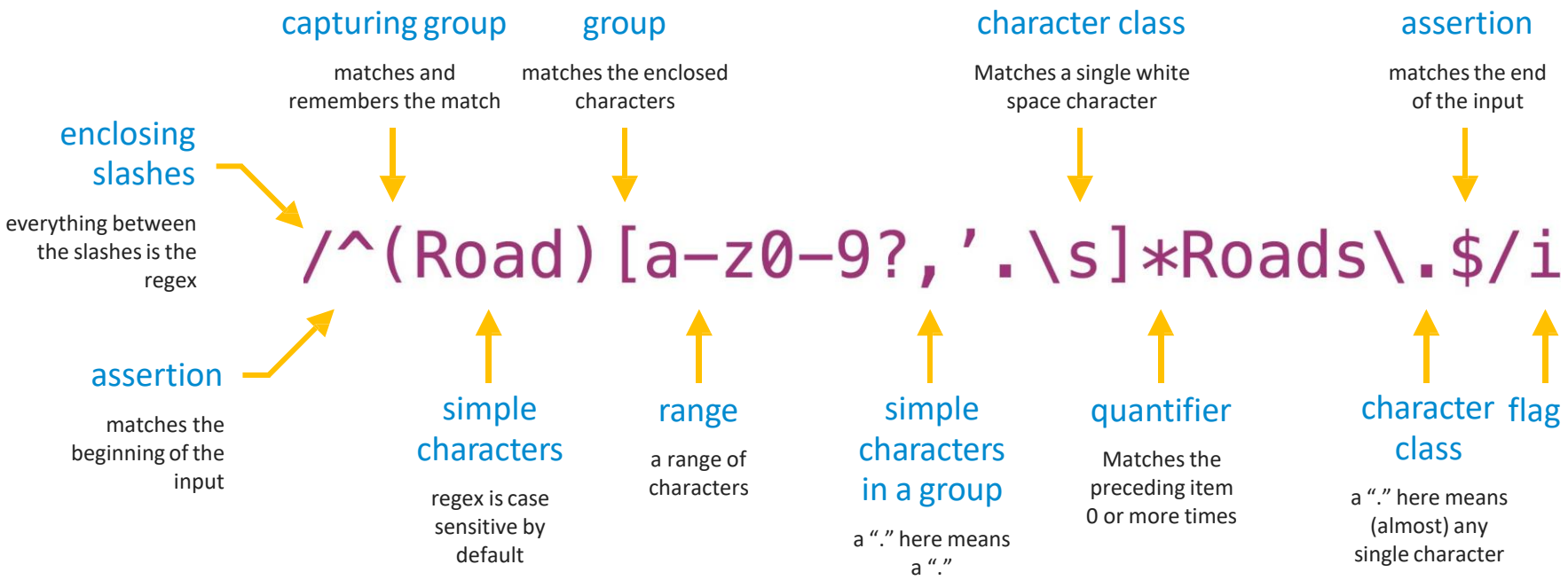
So, there is a simple rule: if you need to create the regex in runtime: you must use the constructor (there is no other way).

In every other case, use the regex literal. Usually, in these cases, you must.

the regex is created runtime

```
> let part = 3;
  let timeMachine = part === 3 ? "train" : "delorean";
  let docExpression = new RegExp(timeMachine, "i");
  docExpression.test("Out Of A DeLorean");
< false
```

Parts of a regex



Regex - flags

Regular expressions have six optional flags that allow for functionality like 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.

The most used flags: "i" - ignore case, and "g" - global

```
> new RegExp(/O/, "g");
<· /O/g
```

the flag

without a flag - first match

```
> "Out Of A DeLorean".replace(/O/, "_");
<· "_ut Of A DeLorean"
```

global flag - multiple matches

```
> "Out Of A DeLorean".replace(/O/g, "_");
<· "_ut _f A DeLorean"
```

ignore case flag

```
> "Out Of A DeLorean".replace(/O/ig, "_");
<· "_ut _f A DeL_rean"
```

MDN: Advanced searching with flags

Regex – flag properties


```
> let regex = /^delorean/gyusim;
< undefined
> regex.flags
< "gimsuy"
> regex.global
< true
> regex.ignoreCase
< true
> regex.multiline
< true
> regex.sticky
< true
> regex.unicode
< true
> regex.dotAll
< true
```

m
y
u
s

Regex flags - methods

We have 2 important built-in methods in RexExp: test() and exec()

The tricky part here is that a RegExp object is stateful:

```
> let str = "Roads? Where We're Going, We Don't Need Roads.";
  let regex = /roads/gi;
undefined
> regex.exec(str);
▶ ["Roads", index: 0, input: "Roads? Where We're Going, We Do
  n't Need Roads.", groups: undefined]
> regex.exec(str);
▶ ["Roads", index: 40, input: "Roads? Where We're Going, We Do
  n't Need Roads.", groups: undefined]
> regex.exec(str);
null
```

```
> let str = "Roads? Where We're Going, We Don't Need Roads.";
  let regex = /roads/gi;
undefined
> regex.lastIndex
0
> regex.test(str);
true
> regex.lastIndex
5
> regex.test(str);
true
> regex.lastIndex
45
> regex.test(str);
false
```

Regex flags – string methods

While RexExp does have test() and exec(), the real power in JavaScript comes from that regex can be used in string methods:

match(), matchAll(), replace(), replaceAll(), search(), split()

```
> let str = "Roads? Where We're Going, We Don't Need Roads.";
< undefined
> let match = str.match(/Roads\./);
< undefined
> match
< ["Roads.", index: 40, input: "Roads? Where We're Going, We Don't Need Roads.", groups: undefined]
    0: "Roads."
    groups: undefined
    index: 40
    input: "Roads? Where We're Going, We Don't Need Roads."
    length: 1
  ▶ __proto__: Array(0)
```

while it returns an array, now we have extra fields here

```
> str.replace(/We/, "You");
< "Roads? Where You're Going, We Don't Need Roads."
```

spot the difference!

```
> str.replace(/We/g, "You");
< "Roads? Where You're Going, You Don't Need Roads."
```

Regex - Quantifiers

characters	meaning
x*	matches the preceding item "x" **0 or more times**
x+	matches the preceding item "x" **1 or more times,** equivalent to {1,}
x?	matches the preceding item "x" **0 or 1 times**
x{n}	matches **exactly "n" occurrences**
x{n,}	**matches at least "n" occurrences** of the preceding item "x"
x{n,m}	**matches at least "n" and at most "m" occurrences** of the preceding item "x"

```
> let str = "Roads? Where We're Going, We Don't Need Roads.";
< undefined
> str.match(/Roads*/g);
< ▶ (2) ["Roads", "Roads"]
> str.match(/Wh*/g);
< ▶ (3) ["Wh", "W", "W"]
> str.match(/Wh+/g);
< ▶ ["Wh"]
> str.match(/e{1}/g);
< ▶ (7) ["e", "e", "e", "e", "e", "e", "e"]
> str.match(/e{2}/g);
< ▶ ["ee"]
```

MDN: Quantifiers

Regex – Groups and Ranges

characters	meaning
x|y	**matches either** "x" or "y"
[xyz] [a-c]	a character class - **matches any** one of the enclosed characters
[^xyz] [^a-c]	a **negated (complemented)** character class - it matches anything that is not enclosed in the brackets
(x)	**capturing group** - matches *x* and remembers the match

```
> let str = "Roads? Where We're Going, We Don't Need Roads.";
< undefined
> str.match(/Roads|We/g);
< ▶ (4) ["Roads", "We", "We", "Roads"]
> str.match(/[wher ]+/gi);
< ▶ (7) ["R", " Where We", "re ", " We ", " ", "ee", " R"]
> str.match(/[^wher ]+/gi);
< ▶ (7) ["oads?", "'", "Going,", "Don't", "N", "d", "oads."]
```

MDN: Groups and Ranges

Regex – Character Classes

characters	meaning
.	Has one of the following meanings: • Matches any single character • Inside a character class, the dot loses its special meaning and matches a literal dot
\d	Matches any **digit** (Arabic numeral) - equivalent to [0-9]
\D	Matches any character that is **not a digit** (Arabic numeral) - equivalent to [^0-9]
\w	Matches any **alphanumeric character** from the basic Latin alphabet, including the underscore - equivalent to [A-Za-z0-9]
\W	Matches any character that is **not a word character** from the basic Latin alphabet - equivalent to [^A-Za-z0-9_]
\s	Matches a **single white space character**, including space, tab, form feed, line feed, and other Unicode spaces.
\S	Matches a single character **other than white space**.
\	Indicates that the following **character should be "escaped".** It behaves one of two ways: • For characters that are usually treated literally, indicates that the next character should not interpreted literally. • For characters that are usually treated specially, indicates that the next character should interpreted literally.

MDN: Character Classes

Regex – Assertions

characters	meaning
^	matches the beginning of input
$	matches the end of input

```
> let str = "Roads? Where We're Going, We Don't Need Roads.";
<· undefined
> str.match(/^[\S]+/g);
<· ▶ ["Roads?"]
> str.match(/[\S]+$/g);
<· ▶ ["Roads."]
> str.match(/[^\w\s]/g);
<· ▶ (5) ["?", "'", ",", "'", "."]
```

MDN: Assertions

THANK YOU!