Code Fellows

Regular Expressions

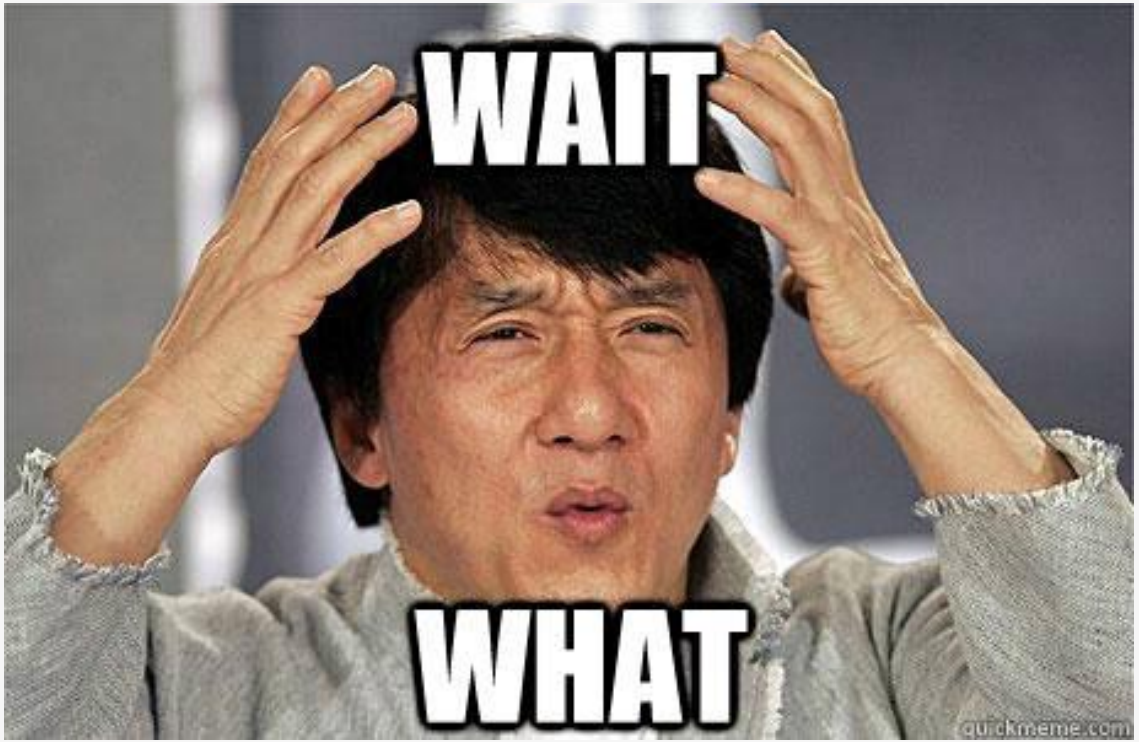
```
/^\(*\d{3}\)*( |-)*\d{3}( |-)*\d{4}$/
```

CODEFELLOWS

TABLE OF CONTENTS

```
/^\(*\d{3}\)*( |-)*\d{3}( |-)*\d{4}$/
```

What does all of that even mean?



WHAT ARE REGULAR EXPRESSIONS?

"Regular expressions are patterns used to match character combinations in strings." ***-MDN***

That's it. That is all they are.

Regular expressions can look bizarre and intimidating at first, but once you learn how to decode them, you'll discover how useful they can be.

String manipulation is a critical skill in software development. Consider how much web content is text: ***strings***. You'll soon learn that the way we move data and web content from one place to another involves one computer passing along to another computer… ***strings***. The code you write… ***strings***.

Strings are everywhere in programming and software development.

And the programmer's single most versatile tool for working with strings: ***regular expressions***.

WHAT ARE REGULAR EXPRESSIONS?

To simplify things, from here on we'll refer to “regular expressions” as simply “regex”. This is a common shorthand you will hear programmers use everywhere, in many programming languages.

Wait… what?

Yes, there are implementations of regex in pretty much all major programming languages. In Code 301 you are learning that despite variations in syntax, many programming languages share numerous features.

Regex is no exception. There will be slight syntax variations from one language to another, but the fundamental idea of what regex does and how we use it remains the same. Wait… what is regex?

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

WHAT ARE REGULAR EXPRESSIONS?

How do programmers use regex?

- Programmers save time with regular expressions
- What is a Regex (Regular Expression)?
- Is it a must for every programmer to learn regular expressions?
- Quora: Why don't more programmers use regular expressions?
- Stack Overflow: Are Regular Expressions a must for programming?

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

WHAT ARE REGULAR EXPRESSIONS?

We're going to work through a series of console-based examples of the string methods `.match()`, `.replace()`, `.search()`, and `.split()`, and see how these methods work when using simple substrings as their inputs, and then gradually replace the substrings with regex patterns of increasing complexity.

You should follow along and play around with the examples.

The simplest way to describe what regex gives us: **wildcards within our comparisons**.

These wildcards can be precise or general, can distinguish letters from numbers from special characters, can find the first match or find all matches in a string, and on and on and on. If you can imagine any kind of match between strings and substrings, there is likely a way to achieve it with regex.

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

EXAMPLES WITH `.match()`

The `match()` method retrieves the matches when matching a string against a regular expression.

For starters, let's try `.match()` by just comparing a string to a simple possible substring:

```
>  'dog'.match('o');
```

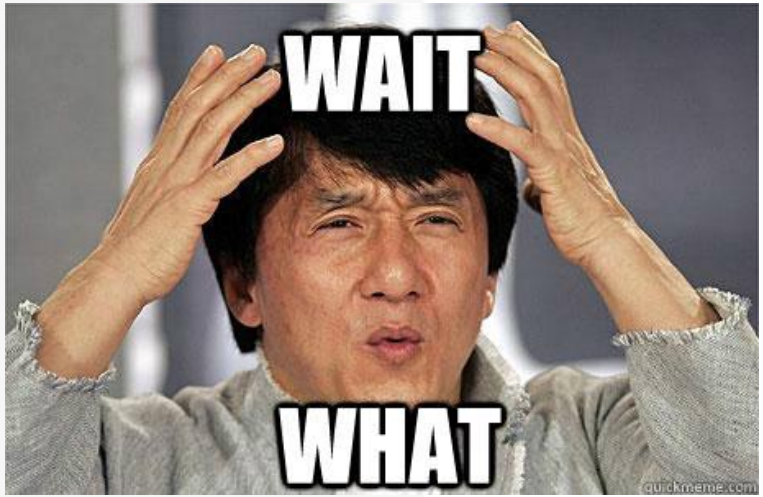
The return value provides a lot of useful information about our attempted match:

```
<• ["o", index: 1, input: "dog", groups: undefined]
```

Take a close look at the output…

...wait… what? Is that an array? It has square brackets…

...but… it looks like there's object properties in there, too!



Yeah. Arrays are objects. Deal with it.

EXAMPLES WITH `.match()`

The **`match()`** method retrieves the matches when matching a string against a regular expression.

Now let's try **`.match()`** for a comparison that will not find any matches:

```
>  'dog'.match('cat');
```

The return value still provides useful information about our attempted match:

```
<• null
```

How is `null` useful? Because it is falsy!

Thus, we can use **`.match()`** to find details about a match between a string and a substring, or, we can simply use truthy/falsy to treat the return value as a boolean and thereby affect control flow logic in **`if()`** and **`while()`** statements, for instance.

Now let's do this with a regex.

(Take a deep breath and crack your knuckles to get ready)

EXAMPLES WITH `.match()`

The `match()` method retrieves the matches when matching a string against a regular expression.

For starters, let's try `.match()` just using a simple regex.

We use slashes to indicate the start and the end of a regex.

```
> 'dog'.match(/o/);
```

The return value provides a lot of useful information about our attempted match:

```
<• ["o", index: 1, input: "dog", groups: undefined]
```

Take a close look at the output… sure looks familiar, doesn't it?

Always remember that the basic concept of regex is simple:

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

We're going to move to some more complex patterns next.

EXAMPLES WITH `.match()`

The `match()` method retrieves the matches when matching a string against a regular expression.

Let's check a string to see if someone mistakenly typed a number in place of a letter, like '0' instead of 'o'.

```
> 'Kookaburra sits in the 01d gum tree'.match(/0/);
<• ["0", index: 23, input: "Kookaburra sits in the 01d gum tree", groups:
undefined]
```

Yep, we got a hit. So let's go in and change the zero to a lowercase “o” to fix it.

But what if we want to check for any other numbers? Is there a shorthand for that?

Of course there is! **`\d`** which stands for “digit. To use this shorthand we need to preface it with a **`\`**

```
> 'Kookaburra sits in the o1d gum tree'.match(/\d/);
<• ["1", index: 24, input: "Kookaburra sits in the o1d gum tree", groups:
undefined]
```

Oooohhh, there's also a “1” in there, too!

EXAMPLES WITH `.match()`

The **`match()`** method retrieves the matches when matching a string against a regular expression.

Now let's revert the string to where it had both numbers in it, and try the shorthand:

```
>  'Kookaburra sits in the 01d gum tree'.match(/\d/);
<• ["0", index: 23, input: "Kookaburra sits in the 01d gum tree", groups: undefined]
```

Hmmm. We know there are two numbers in there, but **`.match()`** only returned the first one that it found.

Surely there is a way to find all of them…?

EXAMPLES WITH `.match()`

The `match()` method retrieves the matches when matching a string against a regular expression.

What we need to do is search the entire string, say, do a ***global*** search, by adding **g** at the end:

```
>  'Kookaburra sits in the 01d gum tree'.match(/\d/g);
<• ["0", "1"]
```

Found them both!

Let's pause for a moment:

Not very long ago, even `/\d/g` would have made your brain hurt.

You can make sense of it now!

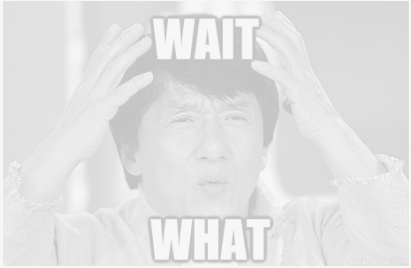
Ultimately, this methodical and piece-by-piece approach is how you will learn regex.

There's no shortcuts. Just gradually build, and give yourself regular practice.

After a quick revisit to our original puzzle, let's do some more.

/^\(*\d{3}\)*(|-)*\d{3}(|-)*\d{4}$/

Let's look at this again, and see how much of it we understand now… and what we can infer.



We're making headway, right? Our confusion is diminishing!

There's still a lot to learn, but recall when you first learned to read...

...wasn't a lot of it letter by letter, then syllable by syllable, then eventually words?

That's how you learn regex.

It's easier to read at first. Writing it is harder. We've all been there.

EXAMPLES WITH `.replace()`

The **`replace()`** method returns a new string with some or all matches of a pattern replaced by a replacement.

Let's dive right in and replace all vowels with a seven.

```
>  'The quick brown fox jumps over the lazy dog'.replace(/[aeiou]/g, 7);
<• 'Th7 q77ck br7wn f7x j7mps 7v7r th7 l7zy d7g'
```

How about only the vowels that are followed by a space, replaced with “XXX ”?

```
>  'The quick brown fox jumps over the lazy dog'.replace(/[aeiou]\s/g, 'XXX ');
<• 'ThXXX quick brown fox jumps over thXXX lazy dog'
```

How about only the letters between “a” and “g”, replaced with a space?

```
>  'The quick brown fox jumps over the lazy dog'.replace(/[a-g]/g, ' ');
<• 'Th  qui k  rown  ox jumps ov r th  l zy  o '
```

That's what I'm talkin' about. Let's do some more.

EXAMPLES WITH `.replace()`

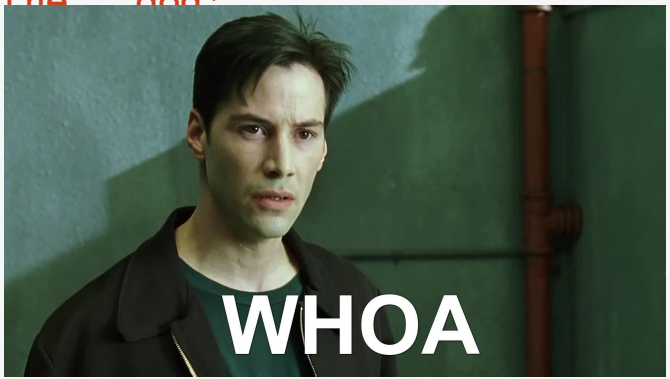
The **`replace()`** method returns a new string with some or all matches of a pattern replaced by a replacement.

Let's replace every letter after a lowercase “o” or “e” and its following letter with a pair of dashes:

```
>  'The quick brown fox jumps over the lazy dog'.replace(/(e|o)[a-z]/g, '--');
<• 'The quick br--n f-- jumps ---- the lazy d--'
```

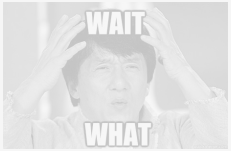
Let's replace every sequence of four consecutive letters with a comma:

```
>  'The quick brown fox jumps over the lazy dog'.replace(/[a-z]{4}/g, ',')
<• 'The ,k ,n fox ,s , the  dog'
```



`/^\(*\d{3}\)*( |-)*\d{3}( |-)*\d{4}$/`

Let's look at this yet again, and see how much of it we understand now… and what we can infer.



Wow, this is actually starting to make sense!

On the prior slide we used this regex: `/(e|o)[a-z]/g`

And this one: `/[a-z]{4}/g`

We've now seen most of what is in the regex above, the one that looked so obtuse when we started. Let's go ahead and parse it completely.

What is this regex doing?

```
/^\(*\d{3}\)*( |-)*\d{3}( |-)*\d{4}$/
```

`/` This is the symbol that opens the regex. Pairs with `/` at the end.

`^` Pairs with `$` to mark the start of a sub-boundary within a regex.

They are not needed here and are merely presented to show their existence.

`\(` The backslash has multiple uses in regex. It can be used as an escape character before a special character, or, when paired with certain characters, forms a token (shorthand) structure (such as `\d`). In this case, it is escaping the parentheses so that the presence of a parentheses character can be checked.

`*` The asterisk matches “one or more or none of the preceding”; in other words, the open parentheses escaped previously may exist or may not exist.

EXAMPLES WITH `.search()`

The **`search()`** method executes a search for a match between a regular expression and a string

`search()` is very similar to the array method **`indexOf()`**, in that it seeks a match, and if it finds one, will return the index of where the match first appears in the string. If there is no match, it returns the value -1.

Let's try a few different **`search()`** examples

```
> const str = 'We at Code Fellows believe that software development skills lead to a better life, community, and world.';
> str.search(/A-Z/g);              // Why did this break?
<• -1
> str.search(/[A-Z]/g);            // Why did this work?
<• 0
> str.search(/F/g);                // Find the first "F"
<• 11
> str[str.search(/[w-z]/g)];       // Returns the first appearance of w,x,y,z
<• w
```

EXAMPLES WITH `.split()`

The split() method splits a String object into an array of strings by separating the string into substrings, using a specified separator string to determine where to make each split.

`split()` is an extremely useful string method. It allows us to turn a string into an array, thereby giving us the ability to apply the usefulness of array methods to the substrings. To turn the array of substrings back into a single string, apply the **`join()`** method to the array, using the same separator.

Let's try a few different **`split()`** examples

```
> const str = 'We at Code Fellows believe that software development skills lead to a better life, community, and world.';
> str.split('e')
<• ["W", " at Cod", " F", "llows b", "li", "v", " that softwar", " d", "v", "lopm", "nt skills l", "ad to a b", "tt", "r lif", ", community, and world."]

> str.split(' ')[6]
<• "software"
```

EXAMPLES WITH `.split()`

The split() method splits a String object into an array of strings by separating the string into substrings, using a specified separator string to determine where to make each split.

`split()` also accepts a regex as the separator, allowing for tremendous versatility.

```
> const str = 'We at Code Fellows believe that software development skills lead to a better life, community, and world.';
> str.split(/be/)
<• ["We at Code Fellows ", "lieve that software development skills lead to a ", "tter life, community, and world."]

> str.split(/e/).reverse().join()
<• ", community, and world.,r lif,tt,ad to a b,nt skills l,lopm,v, d, that softwar,v,li,llows b, F, at Cod,W"
```

SUMMARY

- Regex is versatile and useful.
- Regex can look scary at first, but once you learn how to parse it, quickly becomes readable.
- Learning regex takes time and practice, just like learning any other language.

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

Code Fellows

Regular Expressions

```
/^\(*\d{3}\)*( |-)*\d{3}( |-)*\d{4}$/
```