An Introduction to Regular Expressions (Regex) In JavaScript

Regular Expressions aren't an Alien language. Learn the basics of Regex in JavaScript here.

Brandon Morelli Nov 20

> ***Udemy Black Friday Sale*** *— Thousands of Web Development & Software Development courses are on sale for only $10 for a limited time!* ***Full details and course recommendations can be found here****.*

If Regular Expressions terrify you, you're not alone. To anyone who doesn't understand what they are or how they work, Regex looks like a line of pure gibberish.

But what appears to be gibberish is actually really, really powerful code that can save you a ton of time if you understand how it works. In this article, we'll explore the basics.

Definition

In JavaScript, a regular expression is simply a type of object that is used to match character combinations in strings.

Creating Regex in JS

There are two ways to create a regular expression:

Regular Expression Literal — This method uses slashes (/) to enclose the Regex pattern:

```
var regexLiteral = /cat/;
```

Regular Expression Constructor — This method constructs the Expression for you:

```
var regexConstructor = new RegExp("cat");
```

Each of the above examples reference the same pattern — the character `c`, followed by the character `a`, followed by the character `t`.

As a general rule: If you expect your regular expression to remain constant (unchanging), it is best to use a **regex literal**. If your regular expression is dynamic, it is better to use the **regex constructor** method.

Regular Expression Methods

If you noticed in the definition section above, I mentioned a regular expression is a type of Object.This means there are a number of methods we can use on our regex.

One basic method is `.test()`, which returns a Boolean:

RegExp.prototype.test()

Returns `true`: the string contains a match of the regex pattern
Returns `false`: no match found

Consider the below example. We have two strings and one regular expression. We can use our regex to test if a certain character pattern exists within our strings:

```
const str1 = "the cat says meow";
const str2 = "the dog says bark";

const hasCat = /cat/;

hasCat.test(str1);
// true

hasCat.test(str2);
// false
```

As expected, when we test our first string for `cat`, we get `true`. Also as expected, `str2` does not contain `cat` and thus we get `false`.

A Basic Regex Cheat Sheet

Fortunately, (or Unfortunately depending on your preference), the trick to learning regular expressions is merely just memorizing the common character symbols and groupings.

I've included a reference guide below. If you're serious about understanding regex, when you have some time I encourage you to take 20 minutes and memorize the table below.

Symbols

. — (period) Matches any single character, except for line breaks.
***** — Matches the preceding expression 0 or more times.
+ — Matches the preceding expression 1 or more times.
? — Preceding expression is optional (Matches 0 or 1 times).

^ — Matches the beginning of the string.
$ — Matches the end of the string.

Character groups

\d — Matches any single digit character.
\w — Matches any word character (alphanumeric & underscore).
[XYZ] — Character Set: Matches any single character from the character within the brackets. You can also do a range such as **[A-Z]**
[XYZ]+ — Matches one or more of any of the characters in the set.
[^A-Z] — Inside a character set, the ^ is used for negation. In this example, match anything that is NOT an uppercase letter.

Flags:

There are five optional flags. They can be used separately or together and are placed after the closing slash. Example: `/[A-Z]/g` I'll only be introducing 2 here.
g — Global search
i — case insensitive search

Advanced

(x) — Capturing Parenthesis: Matches x and remembers it so we can use it later.
(?:x) — Non-capturing Parenthesis: Matches x and does not remembers it.
x(?=y) — Lookahead: Matches x only if it is followed by y.

More Advanced Regex Examples

Before we jump into a project, lets explore some of the concepts and keys mentioned above.

First, we'll test a string for any numeric digits. To accomplish this we can use `\d`. Take a look at the below code. It will return `true` as long as there is at least one numeric digit in the test string.

```
console.log(/\d/.test('12-34'));
// true
```

As you can see, we have four numeric digits, so our code will return `true`.

Awesome! But what if we want to match the format of our test string? We can use multiple `\d` characters. For example, to match `11`, we would simply use `\d\d`, which represents any two numbers consecutively. Take a look at the below code:

```
console.log(/\d-\d-\d-\d/.test('1-2-3-4'));
// true

console.log(/\d-\d-\d-\d/.test('1-23-4'));
// false
```

You can see we are testing for any sequence of numbers with dashes between them. Our first test passes as that's what our test string contains. Our second test fails.

But what happens if we don't care how may digits are before or after our dashes so long as there is at least one? We can use the `+` to match the `\d` one or more times. Here it is in action:

```
console.log(/\d+-\d+/.test('12-34'));
// true

console.log(/\d+-\d+/.test('1-234'));
// true

console.log(/\d+-\d+/.test('-34'));
// false
```

To simplify things, we can use parenthesis to group expressions together. Lets say we have a cat meowing and we want to match against that meow:

```
console.log(/me+(ow)+w/.test('meeeeowowoww'));
```

```
// true
```

Woah. OK. Lets break that down. There's a lot going on up here.

```
/me+(ow)+w/
```

```
m     => matching a single letter 'm'
e+    => matching the letter 'e' one or more times
(ow)+ => matching the letters 'ow' one or more times
w     => matching the letter 'w' once
```

```
'm' + 'eeee' +'owowow' + 'w'
```

As you can see above, when operators like `+` are used immediately after parenthesis, they affect the entire contents of those parenthesis.

As a final example we'll look at the `?` operator. The question mark makes the preceding character optional. Take a look at the code below:

```
console.log(/cats? says?/i.test('the Cat says meow'));
// true
```

```
console.log(/cats? says?/i.test('the Cats say meow'));
// true
```

Each test passes. This is because we've made the `s`'s in `cat` and `say` optional. You'll notice I also added an `i` to the end of our regex. This is called a flag and makes our search ignore case. That's why `cat` will still match with `Cat`!

Important Considerations:

Regular expressions are wrapped in slashes. They also make use of other various characters like `+`, `?`, etc. If you ever want to search for these characters, you need to escape them with a backslash. Here's how that would look:

```
var slash = /\//;
var qmark = /\?/;
```

As an additional aside, it's important to know that you can sometimes use multiple character sets to match the same thing. Here are two examples:

`\d` is the same as `[0-9]`: Each will match any number (digit)
`\w` is the same as `[A-Za-z0-9_]`: Each will match any single **alphanumeric** character or **underscore**

Project #1: Add spaces in CamelCase:

Alright! It's finally time to start building some stuff!

In this first project we're going to write a function that will take CamelCase as input and add spaces between the words. Here's the example:

```
removeCc('camelCase') // => will return 'camel Case'
```

The first thing we'll need is a function that takes a string as input and returns a new string:

```
function removeCc(str){
  // return new string
}
```

Perfect, now we just need to fill in our `return` line with regex.

To do this, we'll first search for all capital letters in a string using a character set search and a global modifier:

```
/[A-Z]/g
```

Awesome. The above code will match the C in `camel`**`C`**`ase`. But now, how do we add a space prior to the **`C`**?

For this, we need to use **Capturing Parenthesis**! In Regex, capturing parenthesis allow us to match a value, and remember it. This allows us to use that saved value later. Here's what capturing parenthesis look like:

```
// Capturing Parenthesis
/([A-Z])/

// Access Saved Value
$1
```

Above, you'll see we use `$1` to access the captured value. As an aside, if we had two sets of capturing parenthesis, we would use `$1` and `$2` to reference the captured values in order from left to right. You can use capturing parenthesis as many times as you need.

Note that you don't have to capture the value in parenthesis. You can simply choose to not use it, or you can use non-capturing parenthesis like this: `(?:x)` In this example, the `x` is matched, but is not saved.

Back to the project. There is a string method that we can use to implement the capturing parenthesis — `.replace()`! To use this, our search value will be any capital letter. Our second argument, the replacement value, will be the saved value:

```
function removeCc(str){
  return str.replace(/([A-Z])/g, '$1');
}
```

We're close! But it doesn't quite work yet. Take a look at the code again. We're capturing our capital letter, then simply replacing it with the same captured letter. We need to add in our space. To do this is simple enough, we just add a space before the `$1` variable. The result is a space after each capital letter. Here's what our final code looks like:

```
function removeCc(str){
  return str.replace(/([A-Z])/g, ' $1');
}
```

```
removeCc('camelCase') // 'camel Case'
removeCc('helloWorldItIsMe') // 'hello World It Is Me'
```

Project #2: Remove Capital Letters:

Lets be honest, project #1 only fixed half of the problem. Now we're left with a string that has a bunch of capital letters that shouldn't be there.

In this project, we'll remove the capital letters. Take a moment and try to figure this out on your own before reading on. If you don't get it, don't worry! It's a little complex…

Ok, the first thing we need to do is select all of the capital letters in our string. This is identical to our previous code:

```
/[A-Z]/g
```

We'll also be using the replace method again, but this time we need some new code in our replace method:

```
function lowerCase(str){
  return str.replace(/[A-Z]/g, ???);
}
```

The cool thing about `replace()` is that we specify **a function** as the second parameter. This function will then be invoked after the match has been preformed and the result of the function will be used as the replacement string.

If we also use the global flag, the function will be invoked for each match in the string! With this in mind, we can use the string method `toLowerCase()` to transform our strings to the correct case. Here's what the final code looks like:

```
function lowerCase(str){
  return str.replace(/[A-Z]/g, u => u.toLowerCase());
}
```

```
lowerCase('camel Case') // 'camel case'
lowerCase('hello World It Is Me') // 'hello world it is me'
```

Project #3: Capitalize First Letter:

For the final project, we're going to capitalize the first letter in our string. Here's what should happen:

```
capitalize('camel case') // => should return 'Camel case'
```

Once again we're going to use the `replace()` method. This time however, we only need to search for the very first character in our string. To do this, we will use the `^` character.

As an aside, recall this example from earlier:

```
console.log(/cat/.test('the cat says meow'));
// true
```

When we add in the `^` character, the function will no longer returns true. This is because `cat` is not at the beginning of the string:

```
console.log(/^cat/.test('the cat says meow'));
// false
```

We want our `^` to apply to any lowercase character at the beginning of our string, so we'll add it directly before our character set `[a-z]`. This will target only the first character if it is a lowercase letter.

```
/^[a-z]/
```

We also wont be using the global modifier as we only want one match. From here, all we need to do is return that character as an upper case character and we're done:

```
function capitalize(str){
  return str.replace(/^[a-z]/, u => u.toUpperCase());
}

capitalize('camel case') // 'Camel case'
capitalize('hello world it is me') // 'Hello world it is me'
```

Putting it all together:

We now have all of the pieces to turn a camelCase string into a spaced string with a capital letter at the beginning. Here's what that might look like:

```
function removeCc(str){
  return str.replace(/([A-Z])/g, ' $1');
}

function lowerCase(str){
  return str.replace(/[A-Z]/g, u => u.toLowerCase());
}

function capitalize(str){
  return str.replace(/^[a-z]/, u => u.toUpperCase());
}

capitalize(lowerCase(removeCc('camelCaseIsFun')));
// "Camel case is fun"
```

Awesome! Now it's your turn. Can you combine the three functions above into a single function that turns camelCase into a normal sentence? What about adding a period onto the end of the string?

Post your solutions in the comments!

Closing Notes:

Thanks for reading, and hopefully this was helpful!

If you're ready to finally learn Web Development, check out **The Ultimate**

Guide to Learning Full Stack Web Development in 6 months.

I publish 4 articles on web development each week. Please consider **entering your email here** if you'd like to be added to my once-weekly email list, or follow me on **Twitter**.

If this post was helpful, please click the clap 👏button below a few times to show your support! ⬇⬇

This article was originally published on 4/18/17 and has been edited for clarity and republished.