Lesson | Wednesday

Introduction to Programming (/introduction-to-programming) / Arrays and Looping (/introduction-to-programming/arrays-and-looping) / Further Exploration: Introduction to Regular Expressions

Text | Cheat sheet

Important Note: You aren't required to use regular expressions on any independent projects — and you should prioritize looping during the classwork for this section. **If you need to focus on understanding looping, skip this lesson for now.** It's fine to come back later in this section — or even in future sections — once you're ready to start using regular expressions or if you want to use this lesson as a reference. However, many of the problems in this section can also be solved with regex. If you have time, try refactoring a problem you solve with a loop to use a regular expression instead.

An essential feature of many programming languages, including JavaScript, is the ability to use **regular expressions**. A regular expression (also known as a regex) is a set of characters we can use to find patterns in a string. All the languages we teach at Epicodus, including Ruby and C#, use regular expressions.

While the exact syntax of a regular expression varies somewhat from language to language, the syntax is generally more similar than not. For that reason, once you learn how to use regular

expressions in one language, you'll be well on your way to learning how to use them in other languages, too.

Loops Versus Regular Expressions

Let's start with an example. We'll solve a basic problem with a loop and then demonstrate how we can solve the same problem with a regular expression. Let's say we want to take all the vowels in a string and X them out. Here's how we could do this with a loop:

```
> const vowels = ["a", "i", "e", "o", "u"];
> const string = "Replace all of the vowels with an X, plea
se!";
> const stringArray = string.split("");
> let xArray = [];
> stringArray.forEach(function(letter) {
  if (vowels.includes(letter)) {
    xArray.push("x");
  } else {
    xArray.push(letter);
  }
});
> const finalString = xArray.join("");
> finalString;
"Rxplxcx xll xf thx vxwxls wxth xn X, plxxsx!"
```

As always, try inputting this code in the DevTools console. In this code, we have to split a string and then iterate through it, checking to see if each letter is a vowel or not. If `vowels.includes(letter)`, then we push `'"x"`. Otherwise, we just push the letter.

By the way, the `Array.prototype.includes()` method is super useful — and you may find yourself using it to solve a problem in the classwork. Specifically, it checks to see if an array includes a certain value. In this case, it's a great way to see if the letter is a vowel.

Now let's solve this same problem with a regular expression instead:

```
> const string = "Replace all of the vowels with an X, plea
se!";
> const finalString = string.replace(/[aieou]/gi, "x");
> finalString;
"Rxplxcx xll xf thx vxwxls wxth xn X, plxxsx!"
```

Wow, that's a lot less code — and no loop is needed!

Basic Regular Expressions

In the example above, we use the JavaScript method `String.prototype.replace()`, which takes two arguments. The first argument is the characters (or pattern) that should be replaced while the second is what the characters should be replaced with. Here, we pass in a regular expression (a pattern) as the first argument while the second argument is the letter x. We could also pass in a string into this method as the first argument as well. For instance, we could do the following:

```
> const finalString = string.replace("a", "x");
> finalString;
"Replxce all of the vowels with an X, please!"
```

The difference here is that only a single character is being replaced (the letter "a"), which isn't quite what we want. This method is much more powerful if we use it with a regular expression than with a string.

Now let's take a closer look at the regex above:

```
/[aieou]/gi
```

The part between the slashes `/ /` is the regular expression itself. The part after the final slash includes any flags we want to use with the regular expression.

Regex Flags

We'll start by looking at the flags because they are the easiest part of a regular expression to understand. The `g` and `i` flags are the most common flags we'll use in regular expressions. In fact, most of the time, they'll be the *only* flags we'll ever need.

- `g` means **global**. If we omit this flag, our regular expression will just change the *first* instance of the match in the string, not all of them. Here's what `finalString` is *without* the global flag:

```
"Rxplace all of the vowels with an X, please!"
```

As we can see, just the first vowel was replaced. Since we'll often want to change every instance of a pattern in a string, we'll need to use the `g` flag regularly.

- `i` means **case-insensitive**. If we didn't add this flag, our regex would only change lower-cased vowels — because that's all we've specified in the regular expression itself. Here's an example where we use this regular expression without the `i` flag:

```
> const newString = "Abracadabra!";
> newString.replace(/[aieou]/g, "x");
"Abrxcxdxbrx!"
```

Here, we can see every vowel was replaced — except for the capitalized vowel at the beginning. So the `i` flag is very important to know about, too.

These are the two most commonly used flags and the only ones we really need to know about right now. The other ones are a bit more obscure and generally used with more complex regex — which can get *very* complicated.

Regex Patterns

Now let's look at the pattern in our regular expression:

```
/[aieou]/
```

As we mentioned before, a regex is enclosed in slashes. (The flags aren't part of the pattern itself — they come after the slashes.) The simplest regex is just a pattern with no additional characters. Here's an example:

```
> "The gray cat".replace(/cat/, "dog");
"The gray dog"
```

As we can see, this regex takes the whole pattern, which is the string `"cat"`, and replaces it. It's exactly the same as if we just passed a string in instead.

Regex Groups and Ranges

When we add `[ ]` to a regular expression, it denotes a set or group of characters. So instead of matching the exact pattern `"aieou"` (not at all what we want), our regex will match any of the characters in that group. That's why our example above has the pattern `/[aieou]/`.

Let's say we want to replace *all* characters that are numbers. We could do the following:

```
> const string = "Jasmine's secret code is 13249.";
> string.replace(/\d/g, "x");
"Jasmine's secret code is xxxxx."
```

As we can see, our regex is starting to look a little weirder. We start with two `/ /`. Inside of that, we use `\d`, which means any number. Finally, we use the `g` flag to denote that all numbers in the string should be replaced with an x. If we wanted to replace all characters that are *not* numbers, we'd use `\D` instead. Other common characters include `\w`, which matches any alphanumeric character (letters and numbers) and `\W`, which matches any non-alphanumeric character.

By the way, there are a lot of ways to do the same thing with regular expressions. We can replace Jasmine's secret code using a group with a range like this:

```
> const string = "Jasmine's secret code is 13249.";
> string.replace(/[0-9]/g, "x");
"Jasmine's secret code is xxxxx."
```

The dash (`-`) indicates the range zero to nine — and we need to put it inside the square brackets `[ ]` because this is a group of characters. We can also do `[A-Z]` (for uppercase letters) and `[a-z]` (for lowercase letters). Essentially, `\d` means the same thing as `[0-9]`.

What if we wanted to take our string and replace all the characters that are *not* vowels instead? We can use the `^` character, which means *not* this pattern.

```
const string = "Replace everything that's not a vowel, plea
se!";
string.replace(/[^aeiou]/gi, "x");
> "xexxaxexexexxxxixxxxxaxxxxxoxxaxxoxexxxxxeaxex"
```

This replaced everything that's not a vowel — including the spaces and punctuation!

Regex Characters

A regex character represents certain symbols. We've already covered a few and we'll add a few more:

- `\d` : Numbers
- `\D` : Not numbers
- `\w` : Matches any alphanumeric character (including underscores) — so numbers and letters
- `\W` : Matches any character that's not a number, letter, or underscore
- `\s` : Matches a whitespace character
- `\S` : Matches any non-whitespace character
- `.` : Any single character (wildcard)

Don't worry about memorizing these — just be aware that these characters exist and you can always look at documentation when you actually need them.

Let's go into greater detail regarding the `.`, which represents a character wildcard (unless it's contained inside `[ ]` in which case, it's a period `.` character that we'd want to match in the pattern). We can put `.` before or after other characters to be more flexible in our search for patterns.

Here's an example:

```
> const string = "cat, hat, tilt, colt";
> string.replace(/.t/g, "");
"c, h,i, co"
```

So what happens here? Every time the letter `t` is preceded by another character, it's replaced by empty space. Be careful, though! That includes not just `at` and `lt` but also `t` (with a space) as well. If you wanted it to not include spaces, that would involve a little tinkering.

Quantifiers

Quantifiers allow us to match a specific number of characters. Here are a few that are very useful:

- `+` : Match the preceding character one or more times
- `*` : Match the preceding character zero or more times
- `?` : Match the preceding character zero or one times
- `{ }` : Match a pattern a specified number of times

As we can see, there are some small distinctions in these quantifiers but they can make a big difference. Let's look at some examples.

We can use the `+` to denote a character matching the preceding character 1 or more times. Let's say we have a letter full of exclamation points:

```
> const letter = "Hi there!!!!! How's life?!! It's good her
e!!!!!!!!";
```

It's too many exclamation points! We just want one exclamation point per sentence. We can't just replace them all since we want to keep the first one. We also can't specify an exact pattern like `!!!` because there are differing numbers of exclamation points.

We could do this instead:

```
> letter.replace(/!+/g, "!");
"Hi there! How's life?! It's good here!"
```

The `+` symbol found every pattern in the string with *at least* one `!` and then the method replaced them with a single `!`

While the `+` symbol denotes a character needs to match the preceding character one or more times, the `*` symbol denotes that a character needs to match it *zero* or more times. So it's a little more flexible. It will have a strange effect on the `letter` string above, though:

```
> letter.replace(/!*/g, "!");
"!H!i! !t!h!e!r!e!! !H!o!w!'!s! !l!i!f!e!?!! !I!t!'!s! !g!
o!o!d! !h!e!r!e!!"
```

Every character represents a match. `!` is a match but so is no `!` at all. So it's important to be careful with `*` as it might return unintended matches. This example should illustrate how a tiny little symbol can make all the difference between a successful regex and one that does something completely unintended.

The `*` symbol is much more useful in combination with other characters. Here's an example. Let's say you have a series of old customer service tickets that you want to update. The old tickets begin with X, XY, or XYY. You want to change them to the new system, which begins with AB.

We could do this:

```
> const oldTickets = "X14325, XY15302, XYY5321";
> oldTickets.replace(/XY*/g, "AB");
"AB14325, AB15302, AB5321"
```

Our pattern states "find every X along with every Y that comes directly after it." Then our method replaces this pattern with `"AB"`. For that reason, it replaces `X`, `XY`, and `XYY`. It would replace `XYYYYYYYY`, too.

Finally, the `?` quantifier states that the preceding pattern must match zero or one time. We'll include an example later in this lesson.

There's no need to memorize any of these quantifiers right now (that's what documentation is for), but it's important to be aware that these very subtle distinctions can help us solve a lot of different coding problems.

Here's another quantifier that can be very useful. If we use `{ }`, it denotes the number of characters that can be in a pattern. For instance, let's say we have a string full of numbers that represent scientific data. We know that any numbers that have seven digits are outliers and should be thrown out. (We'll just replace them with zeros here.)

```
> const dataString = "342356, 2345, 4235235, 123, 43534";
> dataString.replace(/\d{7}/g, "0");
'342356, 2345, 0, 123, 43534'
```

This states that patterns with numbers (`\d`) of length `{7}` should be matched.

We could also do matches of different lengths as well:

- `{x}` : `x` is the exact length of characters that should be matched
- `{x,y}` : The match should be *at least* `x` matching characters and *at most* `y` matching characters
- `{x,}` : The match should be at least `x` matching characters with no upper limit

So let's say we want to throw out all numbers that are at least five characters and replace them with `0` :

```
> const dataString = "342356, 2345, 4235235, 123, 43534";
> dataString.replace(/\d{5,}/g, "0");
"0, 2345, 0, 123, 0"
```

As we can see, we use `{5,}` to denote that numbers with `5` or more characters are replaced.

Be careful, though. Look what happens when we forget the comma:

```
> const dataString = "342356, 2345, 4235235, 123, 43534";
> dataString.replace(/\d{5}/g, "0");
"06, 2345, 035, 123, 0"
```

The first number was originally six digits — so this regex pattern matches *exactly* five of these digits and then `String.prototype.replace()` replaces them with a zero. So instead of just taking numbers that are five digits and replacing them with `0` , it will take any cluster of five digits and turn them to zero. For instance:

```
> "23423423423423423423".replace(/\d{5}/g, "0");
"0000"
```

Here, the number being transformed is twenty digits — it matches the pattern four times so it's turned into four zeros. As we can see, this is very different from what happens with the comma.

Other Helpful Regex Symbols

Let's look at a few other helpful regex symbols.

We can use the pipe `|` to represent either/or:

```
> const string = "I see a gray cat and a black dog.";
> string.replace(/cat|dog/g, "bird");
"I see a gray bird and a black bird."
```

We specify either `cat` or `dog` with the pipe `|`. Note that we need to use the `g` flag to change both patterns in the string to `"bird"`.

Finally, it can be very useful to add a pattern boundary. For example, what if we want to find a specific pattern such as `cat` but not when it occurs in words like `cattle` or `cathedral`? We need to denote a pattern boundary, which we can do with `\b`. Let's do a string replacement with and without pattern boundaries.

```
> const string = "I have a cat but I have no cattle. I'd ne
ver take my cat to the cathedral. When my cat hangs out wit
h other cats, they all scatter.";
> string.replace(/cat/g, "dog");
"I have a dog but I have no dogtle. I'd never take my dog t
o the doghedral. When my dog hangs out with other dogs, the
y all sdogter."
```

That's clearly not what we want. Now let's add a pattern boundary both before *and* after so we get exact matches only:

```
> string.replace(/\bcat\b/g, "dog");
"I have a dog but I have no cattle. I'd never take my dog t
o the cathedral. When my dog hangs out with other cats, the
y all scatter."
```

Much better, but there's a gotcha here. Only the patterns that are an exact match are changed. But what about `"cats"`, which isn't an exact match with `"cat"`? We need to change that, too.

This gives us an opportunity to use the `?` we discussed earlier:

```
> string.replace(/\bcats?\b/g, "dog");
"I have a dog but I have no cattle. I'd never take my dog t
o the cathedral. When my dog hangs out with other dog, they
all scatter."
```

As we can see, we can use the `?` after the `s` in our regular expression to denote that the pattern should match `"cat"` (which has no `"s"`) or `"cats"` (which has one `"s"`). Since the `?` is only for zero or one characters, it's exactly what we need.

The string still isn't perfect — the last `"dog"` isn't plural. We can use a regular expression to just change part of the string but that level of complexity is beyond the scope of this lesson. For your current projects, it's perfectly fine to use multiple regex statements (such as one for `cat` and one for `cats`) instead of trying to hunt down the perfect regex that does everything.

Documentation

Check out Mozilla's Regular expression syntax cheatsheet (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions/Cheatsheet) for more information on characters, quantifiers, groups and other

regex symbols. Be warned — there's a *lot* of stuff there. In general, it's good to know some of the basic symbols and then look at documentation when you need something more complex. Also, when you need a *really* complex regex, a well-written Google search will usually lead to an answer in Stack Overflow or elsewhere.

Also, check out the MDN guide to regular expressions (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions) documentation for more information.

If you're interested in practicing regular expressions, check out Regex Crossword (https://regexcrossword.com/), a fun site for learning about regular expressions.

Finally, it's very common to use regex generators that make it easier to get the regex we need to get the job done. A quick Google search will reveal many out there! Here's just a few to optionally check out:

- https://regexr.com/ (https://regexr.com/)
- https://regex-generator.olafneumann.org/ (https://regex-generator.olafneumann.org/)
- https://regex101.com/ (https://regex101.com/)

JavaScript Methods that Use Regex

So far, we've only discussed the `String.prototype.replace()` method. There are several other useful ways to use regular expressions, too, including some very important methods.

First, we can save regular expressions in a variable just like anything else. For instance, we can do this:

```
const justVowels = /[aieou]/;
```

There are also `RegExp` objects as well, but we we won't spend time on it. If you want to explore it on your own, check out MDN's reference page on `RegExp` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp).

`RegExp.prototype.test()`

This method checks whether a string has a specified pattern and returns a boolean. Note that it's not a `String.prototype` method — the receiver (the thing the method is called on) needs to be a regular expression. Here's how we'd use it:

```
> const string = "Jasmine has a cat.";
> /cat/.test(string);
true
> /dog/.test(string);
false
```

This is extremely useful when we just need to know whether a string contains a value. For instance, it's super useful for validating email addresses. We can check that an email address has an `@` as well as a `.com` at the end. If it doesn't, it's not a valid email address.

`String.prototype.match()`

If we want to return an array with all the matches in a pattern, we can use `String.prototype.match()`. Here's an example. Let's say we have a string that includes people's ages. Let's just extract the ages:

```
const string = "Sten is 29. Jayne is 25. Jasmine is 33. Tre
y is 15. Martha is 58. Helen is 83.";
string.match(/\d{1,3}/g);
> ["29", "25", "33", "15", "58", "83"]
```

We use the `String.prototype.match()` method, which takes a regular expression as an argument. Here, our regular expression matches all numbers between one and three digits: `d{1,3}`. Of course, that's an array of strings, so if you want to do some computation with the values, you'd need to change them to numbers.

The three methods we've discussed in this lesson are probably the most useful JavaScript methods that utilize regular expressions but there are others as well.

Summary

We've covered a lot of ground in this lesson — and yes, regular expressions can be overwhelming and confusing, especially at first. Remember, even simple regular expressions can do a lot of heavy lifting in our code — and there's nothing wrong with doing a targeted Google search to get examples of more complex regular expressions if we need to.

Once again, you won't be expected to use regular expressions for this section's independent project (or on any other independent project, either). **You *will* be expected to use a loop for this section's independent project, so make sure you have the hang of those before doing a deep dive into regular expressions.** However, even though it's not essential to have a deep understanding of regular expressions now, they are a very important tool for developers. You will need to have a good handle on them eventually — and you might even find that you can solve a problem in a technical interview with a regular expression as well!

Previous (/introduction-to-programming/arrays-and-looping/practice-pig-latin)
Next (/introduction-to-programming/arrays-and-looping/further-exploration-regular-expressions-with-text-analyzer)

disable dark mode

(http://www.epicodus.com)