Alphabetical Matcher

Now I'd like to show you some of the useful classes and patterns that you might find convenient when you're starting to work with regular expressions.

One of the first ones is the alphabetical matcher inside of brackets instead of typing out all of the letters from "a" to "z". You could just type [a-z]:

```
var source = "The kittens have mittens";
var rhymesKittensMatch = /[a-z]ittens/;
console.log(rhymesKittensMatch.test(source)); //true
```

The lowercase alphabetical matcher also has a companion for uppercase letters:

```
var startsCapitalMatch = /^[A-Z]/;
console.log(startsCapitalMatch.test(source)); //true
```

Matching Lowercase and Capital Letters

One neat thing about regular expressions patterns is that you can also combine these matchers and search for any letter from "a" to "z" lowercase or uppercase by just putting them together inside of square brackets:

```
var onlyLettersMatch = /^[a-zA-Z]$/;
console.log(onlyLettersMatch.test(source)); //false
```

Numerical Matchers

You also might want to match numbers. There is a character class in regular expressions that'll match only digits: `\d`:

```
var source = "There are 100 cats";
var numberMatch = /\d+ cats/;
console.log(numberMatch.test(source)); //true
```

Matching Only Numbers or Anything but Numbers

There's also a regular expression character class for anything that does not matter to it. And that's a backslash followed by a capital "D" (`\D`):

```
var noNumberMatch = /\D+ cats/;
console.log(noNumberMatch.test(source)); //false
```

Match Any Numbers or Letters

There's also a character class that matches any number or letter, uppercase or lowercase and that's the `\w`:

```
var onlyLetterNumberSpace = /^[\w\s]+$/;
console.log(onlyLetterNumberSpace.test(source)); //true
```

`\w` will match any letter or any number, but it won't match spaces, so we add `\s` as well. We get `true` because everything from beginning to end of this string includes only letters, numbers, and spaces.

Quantity Matchers

So these character classes are very convenient for isolating exactly the type of string that you're working with, in case you need to for a regular expression. But in addition, we've been using this plus quantifier in order to say one or more. There's another qualifier that's worth knowing about, and that's the curly braces. With curly braces you can specify any quantity of digits, words, numbers, anything that you can specify as a character class orfor that matter, as a literal character (`{4}`):

```
var source = "There are 100 cats";
var secondSource = "There are 10 cats";
var thirdSource = "There are 10000 cats";
var hundredsMatch = /\d{3} cats/;
console.log(hundredsMatch.test(source)); //true
console.log(hundredsMatch.test(secondSource)); //false
console.log(hundredsMatch.test(thirdSource)); //true
```

So with this regexp we want to match only 3 digits.

On top of that, we can specify a range with curly brackets:

```
var upToThousandsMatch = / \d{3,4} cats/;
console.log(upToThousandsMatch.test(source)); //true
console.log(upToThousandsMatch.test(secondSource)); //false
console.log(upToThousandsMatch.test(thirdSource)); //false
```

This will match either 3 or 4 digits.

Escaping Characters

The backslash is used as an escape and that's the reason that you can create character classes using something like `/d`, and regular expressions won't consider them, thus the letter "d". What this is useful for is matching periods or dollar signs or question marks or other metacharacters that appear in regular expressions that you might actually want to match as part of the string:

```
var source = "The kittens have mittens.";
var endsPeriodMatch = /\.$/;
console.log(endsPeriodMatch.test(source)); //true
```

We escape the dot saying that we want to match the actual period, because otherwise regexp will consider `.` as "any character".