Validation of User Input with Regex

Let's apply the rules to individual fields now so they can validate appropriately

WE'LL COVER THE FOLLOWING

Name

Original rule: Non-empty string of alpha characters

Without regex, the non-empty part can be enforced by `.length` and the `alpha` part can be enforced by checking if the character is within the list `['a', 'b', ..., 'z', 'A', 'B', ..., 'Z']`. With regex, the alpha part is matched by `/[a-zA-Z]/` (don't forget capital letters!), and the nonempty part through the symbol `+` (which means one or more). So that gives us:

```
function isValidName(name) {
  const nameRegex = /^[a-zA-Z]+$/;
  return name.test(nameRegex);
}
```

The `^` means "start" and `$` means "end." Since `test` checks that our regex has a match in any part of the string and we want to validate on the entirety of the string, we need to add those. Otherwise, inputs like `@#$@#$#@ username` would pass.

Username

Original rule: Non-empty string of number, letter, period, or underscore

characters.

This is pretty much the same thing, except we include numbers, `0-9`, the period, `.`, and the underscore, `_`.

```
function isValidUsername(username) {
  const usernameRegex = /^[a-zA-Z0-9._]+$/
  return username.test(usernameRegex);
}
```

Password

Original rule: A string of alphanumeric characters with the following categories.

- *Weak: Less than six-length*
- *Fair: Greater than six-length*
- *Good: Greater than six-length and has a mix of letters and numbers*

This one's a little different in that the return is not a boolean `true`/`false`, and we have a strict number requirement.

Notice we have the following property: **Inputs that satisfy a higher grade category also satisfy the ones below it**.

So a good password is also fair and weak, and a fair password is also weak, technically. We can create 3 different rules and just apply them from the highest grade to lowest, and the first one matched is the category.

As for the strict number requirement, regex has a symbol for that! `{x}` specifies the exact number of occurrences, and `{x,}` specifies at least that number.

The hard part is the "mix of letters and numbers." If we have `/[a-zA-Z0-9]+/`, this enforces that it's a non-empty string with either numbers or letters. The way we do this in pure regex is: `(?=.*[a-zA-Z])(?=.*[0-9])[a-zA-Z0-9]+`

Whoa, right? Briefly, the `?=` is a **lookahead** symbol, which checks characters ahead of the current match, the `.` means "anything," and the `*` means "any number of." So the `(?=.*[a-zA-Z])` means, "for everything in this input, check that there's a letter." The group following it does the same thing with

numbers, and the last section is the actual check for a non-empty string of **either** numbers or letters.

But that's **extremely** messy, and I bet if you were a developer who stumbled upon that piece of code your colleague wrote, you'd have trouble parsing it and be even more cautious making changes. Instead, let's just run multiple, separate validations! It has to pass `/[a-zA-Z]+/`, `/[0-9]+/`, and `/^[a-zA-Z0-9]+$`. The first two check that there are one or more letters and numbers, and the third one checks that the input consists only of numbers or letters.

```
const PASSWORD_CATEGORIES = {
  GOOD: 'password_good',
  FAIR: 'password_fair',
  WEAK: 'password_weak',
}
function getPasswordCategory(password) {
  const hasLettersRegex = /[a-zA-Z]+/
  const hasNumbersRegex = /[0-9]+/
  const hasOnlyLettersAndNumbersRegex = /^[a-zA-Z0-9]{6,}$/

  function isGoodPassword() {
    return hasLettersRegex.test(password) &&
      hasNumbersRegex.test(password) &&
      hasOnlyLettersAndNumbersRegex.test(password);
  }

  function isFairPassword() {
    return hasOnlyLettersAndNumbersRegex.test(password);
  }

  if (isGoodPassword()) {
    return PASSWORD_CATEGORIES.GOOD;
  }
  if (isFairPassword()) {
    return PASSWORD_CATEGORIES.FAIR;
  }
  return PASSWORD_CATEGORIES.WEAK;
}
```

Defining `PASSWORD_CATEGORIES` allows unambiguous communication to this `password` validating interface. Since it's not true or false, we use arbitrary strings, so there's no mixup of what's being returned.

Birthday

Original rule: Numeral characters with enforced character limits of 2 for day and 4 for year.

The month will be a dropdown menu, so we don't need to worry about that.

I think you can do this one yourself!

I think you can do this one yourself.

Q

Which regex will enforce an input with the rule that only allows numbers of length 4?

COMPLETED 0%

1 of 1

Phone Number

Original rule: Numbers with hyphens and parentheses, where parentheses can only surround initial set of numbers and hyphens must be between numbers.

This one is a little more interesting! So far, we've only had a singular group to work with. For this requirement, we need to split things up a little.

In more precise English, the requirement is:

- The input can either begin with a number or an opening parentheses
- If it begins with an opening parenthesis, a closing parenthesis must appear before a hyphen but after one or more numbers
- The input can also include hyphens in between numbers

That's going to be a very messy regex. To simplify, let's split our validation logic for phone numbers into one branch for inputs that have formatting and one that doesn't.

```
FORMATTING_CHARACTERS = ['(', ')', '-'];
```

```
for (const formattingCharacter of FORMATTING_CHARACTERS) {
  if phoneNumber.includes(formattingCharacter) {
    // Do the validation for formatted numbers
  }
}
// Do the validation for non-formatted numbers
```

The validation for non-formatted numbers is trivial. For formats, we don't want to be too strict and not allow users in a country with unique rules.

```
function isValidFormattedNumber(phoneNumber) {
  const regex = /^[0-9(]{1}[0-9)-]+[0-9]$/;
  const hasOpeningParentheses = phoneNumber.includes('(');
  const hasClosingParentheses = phoneNumber.includes(')');
  if (hasOpeningParentheses && !hasClosingParentheses) {
    return false;
  }
  return regex.test(phoneNumber);
}
```

Conditionals are more easily tested outside regexes, and this is much cleaner than whatever the alternative would've been.