LEARN REGULAR EXPRESSIONS BY BUILDING A SPAM FILTER

Introduction:

Regular expressions, often shortened to "regex" or "regexp", are patterns that help programmers match, search, and replace text. Regular expressions are powerful, but can be difficult to understand because they use so many special characters.

In this spam filter project, you'll learn about capture groups, positive lookaheads, negative lookaheads, and other techniques to match any text you want.

Step 1:

To begin the project, use the `.getElementById()` method to retrieve the `#message-input`, `#result`, and `#check-message-btn` elements from the HTML document, and assign them to the variables `messageInput`, `result`, and `checkMessageButton`, respectively.

Step 2:

Attach an event listener to your `checkMessageButton`, listening for the `click` event. Give it an empty callback function.

Step 3:

If the `messageInput` is empty, display an alert to the user with the message `"Please enter a message."`.

Then, exit the function execution.

Step 4:

Create an `isSpam` function using the `const` keyword and arrow syntax. The function should take a single parameter `msg` and implicitly return `false` for now.

Step 5:

Back in your event listener, you need to update the text of the `result` element. You can use a ternary operator to achieve this task.

Here is an example of assigning the result of a ternary operator to an element's text content:

Example Code:

```
el.textContent = condition ? "Use this text if the condition is true"
: "Use this text if the condition is false";
```

After the `if` statement, use a ternary operator to check the truthiness of calling `isSpam()` with `messageInput.value` as the argument. If true, set the `textContent` property on the `result` element to `"Oh no! This looks like a spam message."`. Otherwise, set it to `"This message does not seem to contain any spam."`

Then set the `messageInput` element's `value` property to an empty string.

Step 6:

Your first regular expression will be used to catch help requests. Declare a `helpRegex` variable, and assign it a regular expression that matches the string `please help`.

As a refresher, here is a regular expression to match the string `hello world`:

Example Code:

```
const regex = /hello world/;
```

Step 7:

Regular expressions can take flags to modify their behavior. For instance, the `i` flag can be used to make the expression ignore case, causing it to match `hello`, `HELLO`, and `Hello` for the expression `/hello/`.

Flags are added after the trailing slash. Add the `i` flag to your `helpRegex`.

Step 8:

Strings have a `.match()` method, which accepts a regular expression as an argument and determines if the string matches that expression.

Update your `isSpam()` function to implicitly return the result of calling the `.match()` method on `msg`, passing `helpRegex` as the argument.

Then, try entering some messages on your page and see the result.

Step 9:

Instead of using the `.match()` method, you can use the `.test()` method of a regular expression to test if a string matches the pattern. Unlike `.match()`, `.test()` returns a boolean value indicating whether or not the string matches the pattern.

Update your `isSpam()` function to use the `.test()` method of `helpRegex` to test if `msg` is a match.

Then, try entering some messages on your page and see the result.

Step 10:

The alternate sequence `|` can be used to match either the text on the left or the text on the right of the `|`. For example, the regular expression `/yes|no/` will match either `yes` or `no`.

Update your `helpRegex` to match either `please help` or `assist me`.

Step 11:

Before you start creating additional regular expressions, you need to update your application to check more than one regular expression.

Start by declaring a `denyList` variable. Assign it an array containing your `helpRegex`.

Step 12:

Arrays have a `.some()` method. Like the `.filter()` method, `.some()` accepts a callback function which should take an element of the array as the argument. The `.some()` method will return `true` if the callback function returns `true` for at least one element in the array.

Here is an example of a `.some()` method call to check if any element in the array is an uppercase letter.

Example Code:

```
const arr = ["A", "b", "C"];

arr.some(letter => letter === letter.toUpperCase());
```

Use the `.some()` method to check if testing your `msg` on any of your `denyList` regular expressions returns `true`.

Use `regex` as the parameter for the callback function, for clarity.

Step 13:

The next regular expression you will work on is one that matches mentions of dollar amounts.

Start by declaring a `dollarRegex` variable, and assign it a case-insensitive regular expression that matches the text `dollars`.

Step 14:

Add your `dollarRegex` to the `denyList` array so that you can test the regular expression.

Then try entering a message in your app.

Step 15:

You need to match a number before the text `dollars`. While you could write out `0|1|2` and so on, regular expressions have a feature that makes this easier.

A character class is defined by square brackets, and matches any character within the brackets. For example, `[aeiou]` matches any character in the list `aeiou`. You can also define a range of characters to match using a hyphen. For example, `[a-z]` matches any character from `a` to `z`.

Add a character class to match the digits `0` through `9` to your `dollarRegex` expression - remember the digit must come before the word `dollars`, and there should be a space between the digit and the word.

Step 16:

The dollar value may be more than one digit. To match this, the `+` quantifier can be used - this matches one or more consecutive occurrences. For example, the regular expression `/a+/` matches one or more consecutive `a` characters.

Update your regular expression to match one or more consecutive digits.

Step 17:

Between your digits and your dollars text, you want to catch place values.

Use the `|` token to allow `hundred`, `thousand`, `million`, or `billion` between your digits and dollars.

Step 18:

A capture group is a way to define a part of the expression that should be captured and saved for later reference. You can define a capture group by wrapping a part of your expression in parentheses. For example, `/h(i|ey) camper/` would match either `hi camper` or `hey camper`, and would capture `i` or `ey` in a group.

Turn your place values into a capture group.

Step 19:

Now that you have your capture group, you can mark the entire pattern as an optional match. The `?` quantifier matches zero or one occurrence of the preceding character or group. For example, the regular

expression /colou?r/ matches both color and colour, because the u is optional.

Mark your capture group as optional.

Step 20:

While this expression does match 1 hundred dollars, it will not match 1 hundred dollars, or 10 dollars.

Spam messages can and will find a way to exploit flaws in your detection. Time to improve your regex.

Replace the first literal space with the \s* expression. The \s character class matches whitespace, such as spaces, tabs, and new lines. The * quantifier means "match the previous character 0 or more times".

Replace the second literal space with \s+. The + quantifier means "match the previous character at least one time".

Step 21:

One last thing with this expression. You don't actually need the match value from your capture group, so you can turn it into a non-capturing group. This will allow you to group the characters together without preserving the result.

To create a non-capturing group in a regular expression, you can add ?: after the opening parenthesis of a group. For instance, (?:a|b) will match either a or b, but it will not capture the result.

Update your regular expression to use a non-capturing group.

Step 22:

Your next regular expression will look for strings like `free money`. Start by declaring a `freeRegex` variable and assigning it a regular expression that will match the string `free money`. Remember to make it case-insensitive.

Step 23:

Add your new regular expression to your `denyList` array so you can test it.

Step 24:

Spam messages often use numbers instead of letters to bypass filters. Your regular expression should catch these.

Replace the `e` characters in your regular expression with character classes that match `e` and `3`.

Step 25:

Now update your `o` character to match `o` and `0` (the digit).

Step 26:

Your regex should match whole words, not partial words. That is, you do not want to match `hands-free money management`.

To do this, start by checking for spaces before and after your pattern. You can do this by using the meta character `\s`, which will match spaces, tabs, and line breaks.

Step 27:

If you try entering the message `free money`, you'll notice it doesn't match your expression! This is because `\s` doesn't match the beginning or end of the text.

To match the beginning of the text, you can use the `^` anchor. This asserts that your pattern match starts at the beginning of the full string.

Replace your first `\s` character with a non-capturing group that matches `\s` or `^`.

Step 28:

You still aren't matching `free money` yet, because you need to match the end of the string as well.

Like the `^` anchor, you can use the `$` anchor to match the end of the string.

Update your regular expression to match either the end of the string or a space, like you did for the beginning of the string.

Step 29:

Your next regular expression will match strings like `stock alert`. Declare a `stockRegex` variable and assign it a regular expression that will match the string `stock alert`. Remember to make it case insensitive.

Add it to your `denyList` array as well.

Step 30:

Like your `freeRegex`, update your `stockRegex` to replace the `e` and `o` characters with character classes to match the letter and the corresponding number.

Step 31:

Next update your `s` and `t` characters to also match `5` and `7` respectively.

Step 32:

Character classes can take more than two characters. Replace your `a` character with a character class that matches `a`, `@`, and `4`.

Step 33:

Using the same syntax, update `c` to match `c`, `{`, `[`, and `(`.

Step 34:

Finally, allow your regex to match whole words (like you did with `freeRegex`).

Step 35:

Your final regular expression will look for strings like `dear friend`. Declare a `dearRegex` and assign it a regular expression that will match the string `dear friend`. Remember to make it case insensitive, and add it to your `denyList` array.

Step 36:

To put everything you have learned together, update your `dearRegex` to map the vowels to the corresponding numbers (note that `i` should match `1`, and also match the pipe symbol `|`), and to match whole words.

With that, your spam filter project is complete.