JavaScript Regular Expressions (regex)

Regular Expressions and Common Methods

- **Definition**: Regular Expressions, or Regex, are used to create a "pattern", which you can then use to check against a string, extract text, and more.

```
const regex = /freeCodeCamp/;
```

- **test() Method**: This method accepts a string, which is the string to test for matches against the regular expression. This method will return a boolean if the string matches the regex.

```
const regex = /freeCodeCamp/;

const test = regex.test("e");

console.log(test); // false
```

- **match() Method**: This method accepts a regular expression, although you can also pass a string which will be constructed into a regular expression. The match method returns the match array for the string.

```
const regex = /freeCodeCamp/;

const match = "freeCodeCamp".match(regex);

console.log(match); // ["freeCodeCamp"]
```

- **replace() Method**: This method accepts two arguments: the regular expression to match (or a string), and the string to replace the match with (or a function to run against each match).

```
const regex = /Jessica/;

const str = "Jessica is rly kewl";

const replaced = str.replace(regex, "freeCodeCamp");

console.log(replaced); // "freeCodeCamp is rly kewl"
```

- **replaceAll Method**: This method is used to replace all occurrences of a specified pattern with a new string. This method will throw an error if you give it a regular expression without the global modifier.

```
const text = "I hate JavaScript! I hate programming!";
```

```
const newText = text.replaceAll("hate", "love");

console.log(newText);  // "I love JavaScript! I love programming!"
```

- **matchAll Method**: This method is used to retrieve all matches of a given regular expression in a string, including capturing groups, and returns them as an iterator. An iterator is an object that allows you to go through (or "iterate over") a collection of items.

```
const str = "JavaScript, Python, JavaScript, Swift, JavaScript";

const regex = /JavaScript/g;


const iterator = str.matchAll(regex);


for (let match of iterator) {
  console.log(match[0]); // "JavaScript" for each match
}
```

Regular Expression Modifiers

- **Definition**: Modifiers, often referred to as "flags", modify the behavior of a regular expression.
- **i Flag**: This flag makes a regex ignore case.

```
const regex = /freeCodeCamp/i;

console.log(regex.test("freecodecamp")); // true

console.log(regex.test("FREECODECAMP")); // true
```

- **g Flag**: This flag, or global modifier, allows your regular expression to match a pattern more than once.

```
const regex = /freeCodeCamp/gi;

console.log(regex.test("freeCodeCamp")); // true

console.log(regex.test("freeCodeCamp is great")); // false
```

- **Anchor Definition**: The ^ anchor, at the beginning of the regular expression, says "match the start of the string". The $ anchor, at the end of the regular expression, says "match the end of the string".

```
const start = /^freeCodeCamp/i;

const end = /freeCodeCamp$/i;

console.log(start.test("freecodecamp")); // true

console.log(end.test("freecodecamp")); // true
```

- **m Flag**: Anchors look for the beginning and end of the entire string. But you can make a regex handle multiple lines with the m flag, or the multi-line modifier.flag, or the multi-line modifier.

```
const start = /^freecodecamp/im;

const end = /freecodecamp$/im;

const str = `I love

freecodecamp

it's my favorite

`;

console.log(start.test(str)); // true

console.log(end.test(str)); // true
```

- **d Flag**: This flag expands the information you get in a match object.

```
const regex = /freecodecamp/di;

const string = "we love freecodecamp isn't freecodecamp great?";

console.log(string.match(regex));
```

- **u Flag**: This expands the functionality of a regular expression to allow it to match special unicode characters. The u flag gives you access to special classes like the Extended_Pictographic to match most emoji. There is also a v flag, which further expands the functionality of the unicode matching.

- **y Flag**: The sticky modifier behaves very similarly to the global modifier, but with a few exceptions. The biggest one is that a global regular expression will start from lastIndex and search the entire remainder of the string for another match, but a sticky regular expression will return null and reset the lastIndex to 0 if there is not immediately a match at the previous lastIndex.
- **s Flag**: The single-line modifier allows a wildcard character, represented by a . in regex, to match linebreaks - effectively treating the string as a single line of text.

Character Classes

- **Wildcard .**: Character classes are a special syntax you can use to match sets or subsets of characters. The first character class you should learn is the wild card class. The wild card is represented by a period, or dot, and matches ANY single character EXCEPT line breaks. To allow the wildcard class to match line breaks, remember that you would need to use the s flag.

```
const regex = /a./;
```

- **\d**: This will match all digits (0-9) in a string.

```
const regex = /\d/;
```

- **\w**: This is used to match any word character (a-z0-9_) in a string. A word character is defined as any letter, from a to z, or a number from 0 to 9, or the underscore character.

```
const regex = /\w/;
```

- **\s**: The white-space class \s, represented by a backslash followed by an s. This character class will match any white space, including new lines, spaces, tabs, and special unicode space characters.
- **Negating Special Character Classes**: To negate one of these character classes, instead of using a lowercase letter after the backslash, you can use the uppercase equivalent. The following example does not match a numerical character. Instead, it matches any single character that is NOT a numerical character.

```
const regex = /\D/;
```

- **Custom Character Classes**: You can create custom character classes by placing the character you wish match inside a set of square brackets.

```
const regex = /[abcdf]/;
```

Lookahead and Lookbehind Assertions

- **Definition**: Lookahead and lookbehind assertions allow you to match specific patterns based on the presence or lack of surrounding patterns.
- **Positive Lookahead Assertion**: This assertion will match a pattern when the pattern is followed by another pattern. To construct a positive lookahead, you need to start with the pattern you want to match. Then, use parentheses to wrap the pattern you want to use as your condition. After the opening parenthesis, use ?= to define that pattern as a positive lookahead.

```
const regex = /free(?=code)/i;
```

- **Negative Lookahead Assertion**: This is a type of condition used in regular expressions to check that a certain pattern does not occur ahead in the string.

```
const regex = /free(?!code)/i;
```

- **Positive Lookbehind Assertion**: This assertion will match a pattern only if it is preceded by another specific pattern, without including the preceding pattern in the match.

```
const regex = /(?<=free)code/i;
```

- **Negative Lookbehind Assertion**: This assertion ensures that a pattern is not preceded by another specific pattern. It matches only if the specified pattern is not immediately preceded by the given sequence, without including the preceding sequence in the match.

```
const regex = /(?<!free)code/i;
```

Regex Quantifiers

- **Definition**: Quantifiers in regular expressions specify how many times a pattern (or part of a pattern) should appear. They help control the number of occurrences of characters or groups in a match. The following example is used to match the previous character exactly four times.

```
const regex = /^\d{4}$/;
```

- ***** : Matches 0 or more occurrences of the preceding element.
- **+**: Matches 1 or more occurrences of the preceding element.
- **?**: Matches 0 or 1 occurrence of the preceding element.

- **{n}**: Matches exactly n occurrences of the preceding element.
- **{n,}**: Matches n or more occurrences of the preceding element.
- **{n,m}**: Matches between n and m occurrences of the preceding element.

Capturing Groups and Backreferences

- **Capturing Groups**: A capturing group allows you to "capture" a portion of the matched string to use however you might need. Capturing groups are defined by parentheses containing the pattern to capture, with no leading characters like a lookahead.

```
const regex = /free(code)camp/i;
```

- **Non-Capturing Groups**: A non-capturing group is similar to a capturing group but does not store the matched portion of the string for later use. Non-capturing groups are defined by (?:...).

```
const regex = /free(?:code)camp/i;
```

- **Backreferences**: A backreference in regular expressions refers to a way to reuse a part of the pattern that was matched earlier in the same expression. It allows you to refer to a captured group (a part of the pattern in parentheses) by its number. For example, $1 refers to the first captured group.

```
const regex = /free(co+de)camp/i;

console.log("freecooooooooodecamp".replace(regex, "paid$1world"));
```

- You can use backreferences within the regex itself to match the same text captured by a previous group with a backslash and the capture group number. For example:

```
const regex = /(hello) \1/i;

console.log(regex.test("hello hello"));  // true

console.log(regex.test("hello world"));  // false
```