Regular Expressions

Match Literal Strings using the Test Method:

```
let testStr = "Hello, my name is Kevin.";
let testRegex = /Kevin/;
testRegex.test(testStr); //The test method here returns true.
//Any other forms of Kevin will not match. For example, the regex /Kevin/ will not match k
evin or KEVIN.
```

Match a Literal String with Different Possibilities → You can search for multiple patterns using the `alternation` or `OR` operator: `|`

```
let petString = "James has a pet cat.";
let petRegex = /dog|cat|bird/;
let result = petRegex.test(petString);
```

Ignore Case While Matching: i flag

```
let myString = "freeCodeCamp";
let fccRegex = /freecodecamp/i; // matches any case of freecodecamp string
let result = fccRegex.test(myString);
```

Extract Matches → To use the `.match()` method, apply the method on a string and pass in the regex inside the parentheses.

```
"Hello, World!".match(/Hello/);
let ourStr = "Regular expressions";
let ourRegex = /expressions/;
ourStr.match(ourRegex);
//Here the first match would return ["Hello"] and the second would return ["expressions"].
```

Find More Than the First Match: g flag

```
let repeatRegex = /Repeat/g;
testStr.match(repeatRegex);
//here match returns the value ["Repeat", "Repeat", "Repeat"]

//another example
let twinkleStar = "Twinkle, twinkle, little star";
let starRegex = /Twinkle/ig;
let result = twinkleStar.match(starRegex);
```

Match Anything with Wildcard Period → The wildcard character `.` will match any one character. The wildcard is also called `dot` and `period`

For example, if you wanted to match `hug`, `huh`, `hut`, and `hum`, you can use the regex `/hu./`
to match all four words.

Match Single Character with Multiple Possibilities → You can search for a literal pattern with some flexibility with ***character classes*.** Character classes allow you to define a group of characters you wish to match by placing them inside a square (`[` and `]`) brackets.

For example, you want to match `bag`, `big`, and `bug` but not `bog`. You can create the regex `/b[aiu]g/` to do this. The `[aiu]` **is the character class** that will only match the characters `a`, `i`, or `u`.

Match Numbers and Letters of the Alphabet → Inside a character set, you can define a range of characters to match using a **hyphen character: `-`.**

For example, to match lowercase and uppercase letters `a` through `z` you would use regex

`/[a-z0-9]/ig`**.**

Match Single Characters Not Specified → you could also create a set of characters that you do not want to match. These types of character sets are called ***negated character sets***

To create a negated character set, you place a **caret character (`^`)** after the opening bracket and before the characters you do not want to match.

For example, `/[^aeiou]/gi` matches all characters that are not a vowel. Note that characters like `.`, `!`, `[`, `@`, `/` and white space are matched - the negated vowel character set only excludes the vowel characters.

Match Characters that Occur One or More Times → use the `+` character to check if that is the case. Remember, the character or pattern has to be present consecutively. That is, the character has to repeat one after the other.

For example, `/a+/g` would find one match in `abc` and return `["a"]`. Because of the `+`, it would also find a single match in `aabc` and return `["aa"]`

Match Characters that Occur Zero or More Times → The character to do this is the asterisk or star: `*`

```
let soccerWord = "gooooooooal!";
let gPhrase = "gut feeling";
let oPhrase = "over the moon";
let goRegex = /go*/;
soccerWord.match(goRegex);
gPhrase.match(goRegex);
oPhrase.match(goRegex);
//the three match calls would return the values ["goooooooo"], ["g"], and null.
```

Find Characters with Lazy Matching → In regular expressions, a *greedy* match finds the longest possible part of a string that fits the regex pattern and returns it as a match. The alternative is called a *lazy* match, which finds the smallest possible part of the string that satisfies the regex pattern.

You can apply the regex `/t[a-z]*i/` to the string `"titanic"`. This regex is basically a pattern that starts with `t`, ends with `i`, and has some letters in between.

Regular expressions are by default greedy, so the match would return `["titani"]`. It finds the largest sub-string possible to fit the pattern.

However, you can use the `?` character to change it to lazy matching. `"titanic"` matched against the adjusted regex of `/t[a-z]*?i/` returns `["ti"]`.