**Use export to Share a Code BlockPassed**

Imagine a file called math\_functions.js that contains several functions related to mathematical operations. One of them is stored in a variable, add, that takes in two numbers and returns their sum. You want to use this function in several different JavaScript files. In order to share it with these other files, you first need to export it.

export const add = (x, y) => {

return x + y;

}

The above is a common way to export a single function, but you can achieve the same thing like this:

const add = (x, y) => {

return x + y;

}

export { add };

When you export a variable or function, you can import it in another file and use it without having to rewrite the code. You can export multiple things by repeating the first example for each thing you want to export, or by placing them all in the export statement of the second example, like this:

export { add, subtract };

## Reuse JavaScript Code Using import

import allows you to choose which parts of a file or module to load. In the previous lesson, the examples exported add from the math\_functions.js file. Here's how you can import it to use in another file:

import { add } from './math\_functions.js';

Here, import will find add in math\_functions.js, import just that function for you to use, and ignore the rest. The ./ tells the import to look for the math\_functions.js file in the same folder as the current file. The relative file path (./) and file extension (.js) are required when using import in this way.

You can import more than one item from the file by adding them in the import statement like this:

import { add, subtract } from './math\_functions.js';

## Use \* to Import Everything from a File

Suppose you have a file and you wish to import all of its contents into the current file. This can be done with the import \* as syntax. Here's an example where the contents of a file named math\_functions.js are imported into a file in the same directory:

import \* as myMathModule from "./math\_functions.js";

The above import statement will create an object called myMathModule. This is just a variable name, you can name it anything. The object will contain all of the exports from math\_functions.js in it, so you can access the functions like you would any other object property. Here's how you can use the add and subtract functions that were imported:

myMathModule.add(2,3);

myMathModule.subtract(5,3);

## Create a JavaScript PromisePassed

A promise in JavaScript is exactly what it sounds like - you use it to make a promise to do something, usually asynchronously. When the task completes, you either fulfill your promise or fail to do so. Promise is a constructor function, so you need to use the new keyword to create one. It takes a function, as its argument, with two parameters - resolve and reject. These are methods used to determine the outcome of the promise. The syntax looks like this:

const myPromise = new Promise((resolve, reject) => {

});

## Complete a Promise with resolve and rejectPassed

A promise has three states: pending, fulfilled, and rejected. The promise you created in the last challenge is forever stuck in the pending state because you did not add a way to complete the promise. The resolve and reject parameters given to the promise argument are used to do this. resolve is used when you want your promise to succeed, and reject is used when you want it to fail. These are methods that take an argument, as seen below.

const myPromise = new Promise((resolve, reject) => {

if(condition here) {

resolve("Promise was fulfilled");

} else {

reject("Promise was rejected");

}

});

The example above uses strings for the argument of these functions, but it can really be anything. Often, it might be an object, that you would use data from, to put on your website or elsewhere.

## Handle a Fulfilled Promise with thenPassed

Promises are most useful when you have a process that takes an unknown amount of time in your code (i.e. something asynchronous), often a server request. When you make a server request it takes some amount of time, and after it completes you usually want to do something with the response from the server. This can be achieved by using the then method. The then method is executed immediately after your promise is fulfilled with resolve. Here’s an example:

myPromise.then(result => {

// do something with the result.

});

result comes from the argument given to the resolve method.

## Handle a Rejected Promise with catchPassed

catch is the method used when your promise has been rejected. It is executed immediately after a promise's reject method is called. Here’s the syntax:

myPromise.catch(error => {

// do something with the error.

});

error is the argument passed in to the reject method.

**Note:** the then and catch methods can be chained to the promise declaration if you choose.

const mysql = require(‘mysql’);

function connectToDatabase (config) {

return new Promise(function(resolve, reject) {

const connection = mysql.createConnection(config);

connection.connect(function (err) {

If (err) {

reject(err);

return;

}

resolve(connection);

});

}

connectToDatabase({

host: ‘localhost’,

user: ‘root’,

password: ‘’,

database: ‘chatroom’

}).then(function(conn) {

console.log(‘Connection successful!’);

conn.end();

}).catch(function(err) {

console.log(‘There was an error connecting to the DB!’);

console.log(err.message);

});

## Using the Test Method

Regular expressions are used in programming languages to match parts of strings. You create patterns to help you do that matching.

If you want to find the word "the" in the string "The dog chased the cat", you could use the following regular expression: /the/. Notice that quote marks are not required within the regular expression.

JavaScript has multiple ways to use regexes. One way to test a regex is using the .test() method. The .test() method takes the regex, applies it to a string (which is placed inside the parentheses), and returns true or false if your pattern finds something or not.

let testStr = "freeCodeCamp";

let testRegex = /Code/;

testRegex.test(testStr);

// Returns true

## Match a Literal String with Different Possibilities

Using regexes like /coding/, you can look for the pattern "coding" in another string.

This is powerful to search single strings, but it's limited to only one pattern. You can search for multiple patterns using the alternation or OR operator: |.

This operator matches patterns either before or after it. For example, if you wanted to match "yes" or "no", the regex you want is /yes|no/.

You can also search for more than just two patterns. You can do this by adding more patterns with more OR operators separating them, like /yes|no|maybe/.

## Extract Matches

So far, you have only been checking if a pattern exists or not within a string. You can also extract the actual matches you found with the .match() method.

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

Here's an example:

"Hello, World!".match(/Hello/);

// Returns ["Hello"]

let ourStr = "Regular expressions";

let ourRegex = /expressions/;

ourStr.match(ourRegex);

// Returns ["expressions"]

Note that the .match syntax is the "opposite" of the .test method you have been using thus far:

'string'.match(/regex/);

/regex/.test('string');

## Ignore Case While Matching

Up until now, you've looked at regexes to do literal matches of strings. But sometimes, you might want to also match case differences.

Case (or sometimes letter case) is the difference between uppercase letters and lowercase letters. Examples of uppercase are "A", "B", and "C". Examples of lowercase are "a", "b", and "c".

You can match both cases using what is called a flag. There are other flags but here you'll focus on the flag that ignores case - the i flag. You can use it by appending it to the regex. An example of using this flag is /ignorecase/i. This regex can match the strings "ignorecase", "igNoreCase", and "IgnoreCase".

## Find More Than the First Match

So far, you have only been able to extract or search a pattern once.

let testStr = "Repeat, Repeat, Repeat";

let ourRegex = /Repeat/;

testStr.match(ourRegex);

// Returns ["Repeat"]

To search or extract a pattern more than once, you can use the g flag.

let repeatRegex = /Repeat/g;

testStr.match(repeatRegex);

// Returns ["Repeat", "Repeat", "Repeat"]

## Match Anything with Wildcard Period

Sometimes you won't (or don't need to) know the exact characters in your patterns. Thinking of all words that match, say, a misspelling would take a long time. Luckily, you can save time using the wildcard character: .

The wildcard character . will match any one character. The wildcard is also called dot and period. You can use the wildcard character just like any other character in the regex. For example, if you wanted to match "hug", "huh", "hut", and "hum", you can use the regex /hu./ to match all four words.

let humStr = "I'll hum a song";

let hugStr = "Bear hug";

let huRegex = /hu./;

huRegex.test(humStr); // Returns true

huRegex.test(hugStr); // Returns true

## Match Single Character with Multiple Possibilities

You learned how to match literal patterns (/literal/) and wildcard character (/./). Those are the extremes of regular expressions, where one finds exact matches and the other matches everything. There are options that are a balance between the two extremes.

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 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".

let bigStr = "big";

let bagStr = "bag";

let bugStr = "bug";

let bogStr = "bog";

let bgRegex = /b[aiu]g/;

bigStr.match(bgRegex); // Returns ["big"]

bagStr.match(bgRegex); // Returns ["bag"]

bugStr.match(bgRegex); // Returns ["bug"]

bogStr.match(bgRegex); // Returns null

## Match Letters of the Alphabet

You saw how you can use character sets to specify a group of characters to match, but that's a lot of typing when you need to match a large range of characters (for example, every letter in the alphabet). Fortunately, there is a built-in feature that makes this short and simple.

Inside a character set, you can define a range of characters to match using a hyphen character: -.

For example, to match lowercase letters a through e you would use [a-e].

let catStr = "cat";

let batStr = "bat";

let matStr = "mat";

let bgRegex = /[a-e]at/;

catStr.match(bgRegex); // Returns ["cat"]

batStr.match(bgRegex); // Returns ["bat"]

matStr.match(bgRegex); // Returns null

## Match Single Characters Not Specified

So far, you have created a set of characters that you want to match, but 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

Sometimes, you need to match a character (or group of characters) that appears one or more times in a row. This means it occurs at least once, and may be repeated.

You can 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"].

If it were instead checking the string "abab", it would find two matches and return ["a", "a"] because the a characters are not in a row - there is a b between them. Finally, since there is no "a" in the string "bcd", it wouldn't find a match.

## Match Characters that Occur Zero or More Times

The last challenge used the plus + sign to look for characters that occur one or more times. There's also an option that matches 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); // Returns ["goooooooo"]

gPhrase.match(goRegex); // Returns ["g"]

oPhrase.match(goRegex); // Returns null

## Restrict Possible Usernames

Usernames are used everywhere on the internet. They are what give users a unique identity on their favorite sites.

You need to check all the usernames in a database. Here are some simple rules that users have to follow when creating their username.

1) Usernames can only use alpha-numeric characters.

2) The only numbers in the username have to be at the end. There can be zero or more of them at the end. Username cannot start with the number.

3) Username letters can be lowercase and uppercase.

4) Usernames have to be at least two characters long. A two-character username can only use alphabet letters as characters.

**Example:**

let username = "JackOfAllTrades";

let userCheck = /^[a-z]+([0-9][0-9]+|[a-z])$/i; // Change this line

let result = userCheck.test(username);

**code explanation:**

^ - start of input

[a-z]+ – first character is a letter and it can be 1 or more

[0-9][0-9]+ - ends with 2 or more number

| - or

[a-z] – end with one character

$ - end input

\i – no case sensitive

# [Reuse Patterns Using Capture Group **134**](https://www.freecodecamp.org/learn/javascript-algorithms-and-data-structures/regular-expressions/reuse-patterns-using-capture-groups)

## Hints

### Hint 1

Given code below:

**let** testString = "test test test";

**let** reRegex = /(test)\s\1/;

**let** result = reRegex.test(testString);

result will match only test test because \1 in this example stands for the same text as most recently matched by the 1st capturing group (test).

If we were to literally translate the regex, it would look something like this:

let re = /(test)\s\1/;

let literalRe = /test\stest/;

Both re and literalRe would match the same thing.

### Hint 2

Given the code below:

**let** testString = "test test test";

**let** reRegex = /(test)(\s)\1\2\1/;

**let** result = reRegex.test(testString);

result will match whole test test test because:  
\1 repeats (test)  
\2 repeats (\s)

### Hint 3

The code below:

**let** testString = "test test test test test test";

**let** reRegex = /(test)(\s)\1\2\1/g;

**let** result = reRegex.test(testString);

because we used \g, our Regex doesn’t return after first full match (test test test) and matched all repetitions.

## Use Capture Groups to Search and Replace

Searching is useful. However, you can make searching even more powerful when it also changes (or replaces) the text you match.

You can search and replace text in a string using .replace() on a string. The inputs for .replace() is first the regex pattern you want to search for. The second parameter is the string to replace the match or a function to do something.

let wrongText = "The sky is silver.";

let silverRegex = /silver/;

wrongText.replace(silverRegex, "blue");

// Returns "The sky is blue."

You can also access capture groups in the replacement string with dollar signs ($).

"Code Camp".replace(/(\w+)\s(\w+)/, '$2 $1');

// Returns "Camp Code"