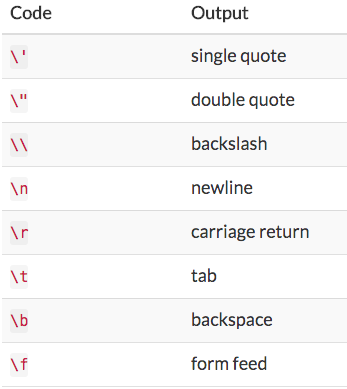
FCC- Javascript

In computer science, *data* is anything that is meaningful to the computer. JavaScript provides seven different *data types* which are undefined, null, boolean, string, symbol, number, and object.

* 

Strings:

* : So if var firstName = "Charles," you can get the value of the first letter of the string by using firstName[0]
* In JavaScript, String values are *immutable*, which means that they cannot be altered once created.
  + Note that this does *not* mean that myStr cannot be changed, just that the individual characters of a *string literal* cannot be changed. The only way to change myStr would be to assign it with a new string, like this

var myStr = "Bob";  
myStr = "Job";

* In order to get the last letter of a string, you can subtract one from the string's length.
* For example, if var firstName = "Charles", you can get the value of the last letter of the string by using firstName[firstName.length - 1]

Arrays

* You can also nest arrays within other arrays, like this: [["Bulls", 23], ["White Sox", 45]]. This is also called a Multi-dimensional Array.
* We can access data inside an array using indexes. Array indexes are written in the same bracket notation that strings use, except that instead of specifying a character, they are specifying an entry in the array.
* Create a variable called myData and set it to equal the first value of myArray using bracket notation.

var myData = myArray[0]

* One way to think of a multi-dimensional array, is as an array of arrays.

**Example**

var arr = [  
  [1,2,3],  
  [4,5,6],  
  [7,8,9],  
  [[10,11,12], 13, 14]  
];  
arr[3]; // equals [[10,11,12], 13, 14]  
arr[3][0]; // equals [10,11,12]  
arr[3][0][1]; // equals 11

* An easy way to append data to the end of an array is via the push()function.
* .push() takes one or more parameters and "pushes" them onto the end of the array.

var arr = [1,2,3];  
arr.push(4);  
// arr is now [1,2,3,4]

* Another way to change the data in an array is with the .pop()function.
* .pop()is used to "pop" a value off of the end of an array. We can store this "popped off" value by assigning it to a variable. In other words, .pop()removes the last element from an array and returns that element.

var threeArr = [1, 4, 6];  
var oneDown = threeArr.pop();  
console.log(oneDown); // Returns 6  
console.log(threeArr); // Returns [1, 4

* pop() always removes the last element of an array. What if you want to remove the first?
* That's where .shift()comes in. It works just like .pop(), except it removes the first element instead of the last.
* Not only can you shift elements off of the beginning of an array, you can also unshift elements to the beginning of an array i.e. add elements in front of the array.
* .unshift()works exactly like .push(), but instead of adding the element at the end of the array, unshift()adds the element at the beginning of the array.

Function

* Variables which are used without the var keyword are automatically created in the global scope. This can create unintended consequences elsewhere in your code or when running a function again. You should always declare your variables with var.
* It is possible to have both local and global variables with the same name. When you do this, the localvariable takes precedence over the globalvariable.
* In JavaScript, you can determine the type of a variable or a value with the typeof operator, as follows:
* typeof 3 // returns 'number'  
  typeof '3' // returns 'string'

Switch Statements and if/else

* Since === returns true or false, we can return the result of the comparison:

function isEqual(a,b) {

return a === b;

}

## Basic JavaScript: Build JavaScript Objects

Objects are similar to arrays, except that instead of using indexes to access and modify their data, you access the data in objects through what are called properties. Objects are useful for storing data in a structured way, and can represent real world objects, like a cat.

Heres a sample cat object:

var cat = {  
  "name": "Whiskers",  
  "legs": 4,  
  "tails": 1,  
  "enemies": ["Water", "Dogs"]  
};

There are two ways to access the properties of an object: dot notation (.) and bracket notation ([]), similar to an array.

Objects can be thought of as a key/value storage, like a dictionary. If you have tabular data, you can use an object to "lookup" values rather than a switch statement or an if/else chain. This is most useful when you know that your input data is limited to a certain range.

## Basic JavaScript: Manipulating Complex Objects

Sometimes you may want to store data in a flexible Data Structure. A JavaScript object is one way to handle flexible data. They allow for arbitrary combinations of strings, numbers, booleans, arrays, functions, and objects.

Here's an example of a complex data structure:

var ourMusic = [  
  {  
    "artist": "Daft Punk",  
    "title": "Homework",  
    "release\_year": 1997,  
    "formats": [   
      "CD",   
      "Cassette",   
      "LP"  
    ],  
    "gold": true  
  }  
];

This is an array which contains one object inside. The object has various pieces of metadata about an album. It also has a nested "formats"array. If you want to add more album records, you can do this by adding records to the top level array.

Objects hold data in a property, which has a key-value format. In the example above, "artist": "Daft Punk"is a property that has a key of "artist" and a value of "Daft Punk".

[JavaScript Object Notation](http://www.json.org/) or JSONis a related data interchange format used to store data.

## Basic JavaScript: Iterate with JavaScript While Loops

The first type of loop we will learn is called a "while" loop because it runs "while" a specified condition is true and stops once that condition is no longer true.

var ourArray = [];  
var i = 0;  
while(i < 5) {  
  ourArray.push(i);  
  i++;  
}

FOR LOOPS

You can run the same code multiple times by using a loop. The most common type of JavaScript loop is called a "for loop" because it runs "for" a specific number of times. For loops are declared with three optional expressions separated by semicolons:

for ([initialization]; [condition]; [final-expression])

## Iterate Through an Array with a For Loop

A common task in JavaScript is to iterate through the contents of an array. One way to do that is with a for loop. This code will output each element of the array arr to the console:

var arr = [10,9,8,7,6];  
for (var i = 0; i < arr.length; i++) {  
   console.log(arr[i]);  
}

Remember that Arrays have zero-based numbering, which means the last index of the array is length - 1. Our condition for this loop is i < arr.length, which stops when I is at length - 1.

## Basic JavaScript: Nesting For Loops

If you have a multi-dimensional array, you can use the same logic as the prior waypoint to loop through both the array and any sub-arrays. Here is an example:

var arr = [  
  [1,2], [3,4], [5,6]  
];  
for (var i=0; i < arr.length; i++) {  
  for (var j=0; j < arr[i].length; j++) {  
    console.log(arr[i][j]);  
  }  
}

This outputs each sub-element in arr one at a time. Note that for the inner loop, we are checking the .length of arr[i], since arr[i] is itself an array.

## Iterate with JavaScript Do...While Loops

You can run the same code multiple times by using a loop.

The next type of loop you will learn is called a "do...while" loop because it first will "do" one pass of the code inside the loop no matter what, and then it runs "while" a specified condition is true and stops once that condition is no longer true. Let's look at an example.

var ourArray = [];  
var i = 0;  
do {  
  ourArray.push(i);  
  i++;  
} while (i < 5);

This behaves just as you would expect with any other type of loop, and the resulting array will look like [0, 1, 2, 3, 4]. However, what makes the do...while different from other loops is how it behaves when the condition fails on the first check. Let's see this in action.

Essentially, a do...while loop ensures that the code inside the loop will run at least once.

Let's try getting a do...while loop to work by pushing values to an array.

Change the while loop in the code to a do...while loop so that the loop will push the number 10 to myArray, and I will be equal to 11 when your code finishes running.

## Generate Random Fractions with JavaScript

Random numbers are useful for creating random behavior.

JavaScript has a Math.random() function that generates a random decimal number between 0(inclusive) and not quite up to 1(exclusive). Thus Math.random()can return a 0 but never quite return a 1

Use another function, Math.floor()to round the number down to its nearest whole number.

Math.floor(Math.random() \* 20); //generates a whole number from 0-19

Instead of generating a random number between zero and a given number like we did before, we can generate a random number that falls within a range of two specific numbers.

To do this, we'll define a minimum number min and a maximum number max.

## Basic JavaScript: Use the parseInt Function

The parseInt() function parses a string and returns an integer. Here's an example:

var a = parseInt("007");

WITH A RADIX

If the first character in the string can't be converted into a number, then it returns NaN

The parseInt() function parses a string and returns an integer. It takes a second argument for the radix, which specifies the base of the number in the string. The radix can be an integer between 2 and 36.

The function call looks like:

parseInt(string, radix);

And here's an example:

var a = parseInt("11", 2);

The radix variable says that "11" is in the binary system, or base 2. This example converts the string "11" to an integer 3.

## Use the Conditional (Ternary) Operator

The conditional operator, also called the ternary operator, can be used as a one line if-else expression.

The syntax is:

condition ? statement-if-true : statement-if-false;

MULTIPLE

function findGreaterOrEqual(a, b) {  
  if(a === b) {  
    return "a and b are equal";  
  }  
  else if(a > b) {  
    return "a is greater";  
  }  
  else {  
    return "b is greater";  
  }  
}

The above function can be re-written using multiple conditional operators:

function findGreaterOrEqual(a, b) {  
  return (a === b) ? "a and b are equal" : (a > b) ? "a is greater" : "b is greater";  
}

## ES6: Explore Differences Between the var and let Keywords

One of the biggest problems with declaring variables with the var keyword is that you can overwrite variable declarations without an error. In a small application, you might not run into this type of problem, but when your code becomes larger, you might accidentally overwrite a variable that you did not intend to overwrite.

Because this behavior does not throw an error, searching and fixing bugs becomes more difficult

A new keyword called let was introduced in ES6 to solve this potential issue with the var keyword. So unlike var, when using let, a variable with the same name can only be declared once.

Note the "use strict". This enables Strict Mode, which catches common coding mistakes and "unsafe" actions. For instance:

## Compare Scopes of the var and let Keywords

When you declare a variable with the var keyword, it is declared globally, or locally if declared inside a function.

The let keyword behaves similarly, but with some extra features. When you declare a variable with the let keyword inside a block, statement, or expression, its scope is limited to that block, statement, or expression.

**By using let you can declare variables in relation to their scope.**

function checkScope() {

"use strict";

let i = "function scope";

if (true) {

let i = "block scope";

console.log("Block scope i is: ", i);

}

console.log("Function scope i is: ", i);

return i;

}

ES6: Declare a Read-Only Variable with the const Keyword

Const has all the awesome features that let has, with the added bonus that variables declared using const are read-only. They are a constant value, which means that once a variable is assigned with const, it cannot be reassigned.

A common practice when naming constants is to use all uppercase letters, with words separated by an underscore. Const SENTENCE = “Gideon is awesome”

The const declaration has many use cases in modern JavaScript.

Some developers prefer to assign all their variables using const by default, unless they know they will need to reassign the value. Only in that case, they use let.

## ES6: Prevent Object Mutation

As seen in the previous challenge, const declaration alone doesn't really protect your data from mutation. To ensure your data doesn't change, JavaScript provides a function Object.freeze to prevent data mutation.

Once the object is frozen, you can no longer add, update, or delete properties from it. Any attempt at changing the object will be rejected without an error.

let obj = {  
  name:"FreeCodeCamp"  
  review:"Awesome"  
};  
Object.freeze(obj);  
obj.review = "bad"; //will be ignored. Mutation not allowed

## Use Arrow Functions to Write Concise Anonymous Functions

In JavaScript, we often don't need to name our functions, especially when passing a function as an argument to another function. Instead, we create inline functions. We don't need to name these functions because we do not reuse them anywhere else.

To achieve this, we often use the following syntax:

const myFunc = function() {  
  const myVar = "value";  
  return myVar;  
}

When there is no function body, and only a return value, arrow function syntax allows you to omit the keyword return as well as the brackets surrounding the code. This helps simplify smaller functions into one-line statements:

const myFunc= () => "value"

This code will still return value by default.

In JS, we often, we often don't name our functions. Callback functions, for example.

Because we often create function just to pass it as argument to some other function, and we might not need to reuse this function anywhere.

To achieve this, we often use the boilerplate

function(){

// function body

return ;

}

ES6 provides you with the syntactic sugar to not having to write this. Instead, you can use arrow function syntax

() => (  
// function body  
return ;  
)

When there is no function body, and only a return value, you can simply write that as a one-liner

```js

() => (/\* return value \*/)

## Set Default Parameters for Your Functions

In order to help us create more flexible functions, ES6 introduces *default parameters*for functions.

Check out this code:

function greeting(name = "Anonymous") {  
  return "Hello " + name;  
}  
console.log(greeting("John")); // Hello John  
console.log(greeting()); // Hello Anonymous

The default parameter kicks in when the argument is not specified (it is undefined). As you can see in the example above, the parameter namewill receive its default value "Anonymous"when you do not provide a value for the parameter. You can add default values for as many parameters as you want.

## Use the Rest Operator with Function Parameters

In order to help us create more flexible functions, ES6 introduces the rest operator for function parameters. With the rest operator, you can create functions that take a variable number of arguments. These arguments are stored in an array that can be accessed later from inside the function.

Rest: Collects multiple elements and condenses into a single array element.

Check out this code:

function howMany(...args) {  
  return "You have passed " + args.length + " arguments.";  
}  
console.log(howMany(0, 1, 2)); // You have passed 3 arguments  
console.log(howMany("string", null, [1, 2, 3], { })); // You have passed 4 arguments.

The rest operator eliminates the need to check the args array and allows us to apply map(), filter()and reduce()on the parameters array.

Spread Operator

<https://www.youtube.com/watch?feature=player_embedded&v=iLx4ma8ZqvQ>

Spread: It is … Allows expressions to be expanded in places where multiple arguments, elements, or

However, the spread operator only works in-place, like in an argument to a function or in an array literal.

-Common uses cases:

* Add elements of an existing array into an arrayPass elements (certification and certtoadd example where individual strings were added instead of array within an array).
* Pass elements of an array as arguments to a function.
* variables are expected. It spreads/expands ana array into its elements

## Use Destructuring Assignment to Assign Variables from Objects

We saw earlier how spread operator can effectively spread, or unpack, the contents of the array. We can do something similar with objects as well. *Destructuring assignment* is special syntax for neatly assigning values taken directly from an object to variables.

Consider the following ES5 code:

var voxel = {x: 3.6, y: 7.4, z: 6.54 };  
var x = voxel.x; // x = 3.6  
var y = voxel.y; // y = 7.4  
var z = voxel.z; // z = 6.54

Here's the same assignment statement with ES6 destructuring syntax:

const {x, y, z } = voxel; // x = 3.6, y = 7.4, z = 6.54

If instead you want to store the values of voxel.xinto a, voxel.yinto b, and voxel.zinto c, you have that freedom as well.

const {x : a, y : b, z : c } = voxel // a = 3.6, b = 7.4, c = 6.54

You may read it as "get the field x and copy the value into a," and so on.

## ES6: Use Destructuring Assignment to Assign Variables from Nested Objects

We can similarly destructure *nested* objects into variables.

Consider the following code:

const a = {  
  start: { x: 5, y: 6},  
  end: { x: 6, y: -9 }  
};  
const { start : { x: startX, y: startY }} = a;  
console.log(startX, startY); // 5, 6

In the example above, the variable start is assigned the value of a.start, which is also an object.

## se Destructuring Assignment to Assign Variables from Arrays

ES6 makes destructuring arrays as easy as destructuring objects.

One key difference between the spread operator and array destructuring is that the spread operator unpacks all contents of an array into a comma-separated list. Consequently, you cannot pick or choose which elements you want to assign to variables.

Destructuring an array lets us do exactly that:

const [a, b] = [1, 2, 3, 4, 5, 6];  
console.log(a, b); // 1, 2

## ES6: Use Destructuring Assignment with the Rest Operator to Reassign Array Elements

In some situations involving array destructuring, we might want to collect the rest of the elements into a separate array.

The result is similar to Array.prototype.slice(), as shown below:

const [a, b, ...arr] = [1, 2, 3, 4, 5, 7];  
console.log(a, b); // 1, 2  
console.log(arr); // [3, 4, 5, 7]

Variables aand btake the first and second values from the array. After that, because of rest operator's presence, arr gets rest of the values in the form of an array.

## ES6: Write Concise Declarative Functions with ES6

When defining functions within objects in ES5, we have to use the keyword functionas follows:

const person = {  
  name: "Taylor",  
  sayHello: function() {  
    return `Hello! My name is ${this.name}.`;  
  }  
};

With ES6, You can remove the functionkeyword and colon altogether when defining functions in objects. Here's an example of this syntax:

const person = {  
  name: "Taylor",  
  sayHello() {  
    return `Hello! My name is ${this.name}.`;  
  }  
};

## ES6: Understand the Differences Between import and require

In the past, the function require()would be used to import the functions and code in external files and modules. While handy, this presents a problem: some files and modules are rather large, and you may only need certain code from those external resources.

ES6 gives us a very handy tool known as import. With it, we can choose which parts of a module or file to load into a given file, saving time and memory.

Consider the following example. Imagine that math\_array\_functionshas about 20 functions, but I only need one, countItems, in my current file. The old require()approach would force me to bring in all 20 functions. With this new importsyntax, I can bring in just the desired function, like so:

import { countItems } from "math\_array\_functions"

A description of the above code:

import { function } from "file\_path\_goes\_here"  
// We can also import variables the same way!

There are a few ways to write an import statement, but the above is a very common use-case.

**Note**  
The whitespace surrounding the function inside the curly braces is a best practice - it makes it easier to read the importstatement.

**Note**  
The lessons in this section handle non-browser features. import, and the statements we introduce in the rest of these lessons, won't work on a browser directly. However, we can use various tools to create code out of this to make it work in browser.

**Note**  
In most cases, the file path requires a ./before it; otherwise, node will look in the node\_modulesdirectory first trying to load it as a dependency.

Add the appropriate import statement that will allow the current file to use the capitalizeStringfunction. The file where this function lives is called "string\_functions", and it is in the same directory as the current file.

## ES6: Use export to Reuse a Code Block

In the previous challenge, you learned about importand how it can be leveraged to import small amounts of code from large files. In order for this to work, though, we must utilize one of the statements that goes with import, known as export. When we want some code - a function, or a variable - to be usable in another file, we must export it in order to import it into another file. Like import, exportis a non-browser feature.

The following is what we refer to as a named export. With this, we can import any code we export into another file with the importsyntax you learned in the last lesson. Here's an example:

const capitalizeString = (string) => {  
  return string.charAt(0).toUpperCase() + string.slice(1);  
}  
export { capitalizeString } //How to export functions.  
export const foo = "bar"; //How to export variables.

## ES6: Use \* to Import Everything from a File

Suppose you have a file that you wish to import all of its contents into the current file. This can be done with the import \* syntax.

Here's an example where the contents of a file named "math\_functions"are imported into a file in the same directory:

import \* as myMathModule from "math\_functions";  
myMathModule.add(2,3);  
myMathModule.subtract(5,3);

And breaking down that code:

import \* as object\_with\_name\_of\_your\_choice from "file\_path\_goes\_here"  
object\_with\_name\_of\_your\_choice.imported\_function

You may use any name following the import \* as portion of the statement. In order to utilize this method, it requires an object that receives the imported values. From here, you will use the dot notation to call your imported values.

## ES6: Create an Export Fallback with export default

In the exportlesson, you learned about the syntax referred to as a named export. This allowed you to make multiple functions and variables available for use in other files.

There is another exportsyntax you need to know, known as export default. Usually you will use this syntax if only one value is being exported from a file. It is also used to create a fallback value for a file or module.

Here is a quick example of export default:

export default function add(x,y) {  
  return x + y;  
}

Note: Since export defaultis used to declare a fallback value for a module or file, you can only have one value be a default export in each module or file. Additionally, you cannot use export default with var, let, or const

## ES6: Import a Default Export

In the last challenge, you learned about export defaultand its uses. It is important to note that, to import a default export, you need to use a different importsyntax.

In the following example, we have a function, add, that is the default export of a file, "math\_functions". Here is how to import it:

import add from "math\_functions";  
add(5,4); //Will return 9

The syntax differs in one key place - the imported value, add, is not surrounded by curly braces, {}. Unlike exported values, the primary method of importing a default export is to simply write the value's name after import.

## Introduction to the Regular Expression Challenges

Regular expressions are special strings that represent a search pattern. Also known as "regex" or "regexp", they help programmers match, search, and replace text. Regular expressions can appear cryptic because a few characters have special meaning. The goal is to combine the symbols and text into a pattern that matches what you want, but only what you want. This section will cover the characters, a few shortcuts, and the common uses for writing regular expressions.

* 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 trueor falseif your pattern finds something or not.

let testStr = "freeCodeCamp";  
let testRegex = /Code/;  
testRegex.test(testStr);  
// Returns true

* 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.
* 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/.
* 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 iflag. 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".

## Regular Expressions: 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"]

**Finding more than the FIrest mAtch**

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

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.

Greedy vs Lazy Match

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"].

## Match Beginning String Patterns

Prior challenges showed that regular expressions can be used to look for a number of matches. They are also used to search for patterns in specific positions in strings.

In an earlier challenge, you used the caretcharacter (^) inside a character setto create a negated character set in the form [^thingsThatWillNotBeMatched]. Outside of a character set, the caretis used to search for patterns at the beginning of strings.

let firstString = "Ricky is first and can be found.";  
let firstRegex = /^Ricky/;  
firstRegex.test(firstString);  
// Returns true  
let notFirst = "You can't find Ricky now.";  
firstRegex.test(notFirst);  
// Returns false

Recall that you use the plus sign + to look for one or more characters and the asterisk \* to look for zero or more characters. These are convenient but sometimes you want to match a certain range of patterns.

You can specify the lower and upper number of patterns with quantity specifiers. Quantity specifiers are used with curly brackets ({and }). You put two numbers between the curly brackets - for the lower and upper number of patterns.

LOOKAHEADS

Lookaheadsare patterns that tell JavaScript to look-ahead in your string to check for patterns further along. This can be useful when you want to search for multiple patterns over the same string.

There are two kinds of lookaheads: positive lookahead and negative lookahead.

A positive lookahead will look to make sure the element in the search pattern is there, but won't actually match it. A positive lookahead is used as (?=...)where the ...is the required part that is not matched.

On the other hand, a negative lookahead will look to make sure the element in the search pattern is not there. A negative lookahead is used as (?!...)where the ...is the pattern that you do not want to be there. The rest of the pattern is returned if the negative lookahead part is not present.

A more practical use of lookaheads is to check two or more patterns in one string. Here is a (naively) simple password checker that looks for between 3 and 6 characters and at least one number:

let password = "abc123";  
let checkPass = /(?=\w{3,6})(?=\D\*\d)/;  
checkPass.test(password); // Returns true

CAPTURE GROUPS

Some patterns you search for will occur multiple times in a string. It is wasteful to manually repeat that regex. There is a better way to specify when you have multiple repeat substrings in your string.

You can search for repeat substrings using capture groups. Parentheses, (and ), are used to find repeat substrings. You put the regex of the pattern that will repeat in between the parentheses.

To specify where that repeat string will appear, you use a backslash (\) and then a number. This number starts at 1 and increases with each additional capture group you use. An example would be \1to match the first group.

The example below matches any word that occurs twice separated by a space:

let repeatStr = "regex regex";  
let repeatRegex = /(\w+)\s\1/;  
repeatRegex.test(repeatStr); // Returns true  
repeatStr.match(repeatRegex); // Returns ["regex regex", "regex"]

Using the .match()method on a string will return an array with the string it matches, along with its capture group.