**Accessing Object Properties with Variables**

Another use of bracket notation on objects is to access a property which is stored as the value of a variable. This can be very useful for iterating through an object's properties or when accessing a lookup table.

Here is an example of using a variable to access a property:

var dogs = {

Fido: "Mutt", Hunter: "Doberman", Snoopie: "Beagle"

};

var myDog = "Hunter";

var myBreed = dogs[myDog];

console.log(myBreed);

The string Doberman would be displayed in the console.

Another way you can use this concept is when the property's name is collected dynamically during the program execution, as follows:

var someObj = {

propName: "John"

};

function propPrefix(str) {

var s = "prop";

return s + str;

}

var someProp = propPrefix("Name");

console.log(someObj[someProp]);

someProp would have a value of the string propName, and the string John would be displayed in the console.

Note that we do *not* use quotes around the variable name when using it to access the property because we are using the *value* of the variable, not the *name*.

Set the playerNumber variable to 16. Then, use the variable to look up the player's name and assign it to player.

// Setup

var testObj = {

12: "Namath",

16: "Montana",

19: "Unitas"

};

// Only change code below this line

var playerNumber=16; // Change this line

var player = testObj[playerNumber]; // Change this line

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Updating Object Properties**

After you've created a JavaScript object, you can update its properties at any time just like you would update any other variable. You can use either dot or bracket notation to update.

For example, let's look at ourDog:

var ourDog = {

"name": "Camper",

"legs": 4,

"tails": 1,

"friends": ["everything!"]

};

Since he's a particularly happy dog, let's change his name to the string Happy Camper. Here's how we update his object's name property: ourDog.name = "Happy Camper"; or ourDog["name"] = "Happy Camper"; Now when we evaluate ourDog.name, instead of getting Camper, we'll get his new name, Happy Camper.

Update the myDog object's name property. Let's change her name from Coder to Happy Coder. You can use either dot or bracket notation.

// Setup

var myDog = {

"name": "Coder",

"legs": 4,

"tails": 1,

"friends": ["freeCodeCamp Campers"]

};

// Only change code below this line

myDog.["name"]="Happy Coder";

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Add New Properties to a JavaScript Object**

Passed

You can add new properties to existing JavaScript objects the same way you would modify them.

Here's how we would add a bark property to ourDog:

ourDog.bark = "bow-wow";

or

ourDog["bark"] = "bow-wow";

Now when we evaluate ourDog.bark, we'll get his bark, bow-wow.

Example:

var ourDog = {

"name": "Camper",

"legs": 4,

"tails": 1,

"friends": ["everything!"]

};

ourDog.bark = "bow-wow";

Add a bark property to myDog and set it to a dog sound, such as "woof". You may use either dot or bracket notation.

var myDog = {

"name": "Happy Coder",

"legs": 4,

"tails": 1,

"friends": ["freeCodeCamp Campers"]

};

myDog["bark"]="woof";

console.log(myDog);

console.log(myDog.bark);

console.log(myDog.["bark"]);

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Delete Properties from a JavaScript Object**

We can also delete properties from objects like this:

delete ourDog.bark;

Example:

var ourDog = {

"name": "Camper",

"legs": 4,

"tails": 1,

"friends": ["everything!"],

"bark": "bow-wow"

};

delete ourDog.bark;

After the last line shown above, ourDog looks like:

{

"name": "Camper",

"legs": 4,

"tails": 1,

"friends": ["everything!"]

}

Delete the tails property from myDog. You may use either dot or bracket notation.

// Setup

var myDog = {

"name": "Happy Coder",

"legs": 4,

"tails": 1,

"friends": ["freeCodeCamp Campers"],

"bark": "woof"

};

// Only change code below this line

delete myDog.tails;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Using Objects for Lookups**

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.

Here is an example of a simple reverse alphabet lookup:

var alpha = {

1:"Z",

2:"Y",

3:"X",

4:"W",

...

24:"C",

25:"B",

26:"A"

};

alpha[2];

alpha[24];

var value = 2;

alpha[value];

alpha[2] is the string Y, alpha[24] is the string C, and alpha[value] is the string Y.

Convert the switch statement into an object called lookup. Use it to look up val and assign the associated string to the result variable.

// Setup

function phoneticLookup(val) {

var result = "";

// Only change code below this line

var lookup={

"alpha":"Adams","bravo":"Boston","charlie":"Chicago","delta":"Denver","echo":"Easy","foxtrot":"Frank"

}

result= lookup[val];

// Only change code above this line

return result;

}

phoneticLookup("charlie");

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Testing Objects for Properties**

Sometimes it is useful to check if the property of a given object exists or not. We can use the .hasOwnProperty(propname) method of objects to determine if that object has the given property name. .hasOwnProperty() returns true or false if the property is found or not.

**Example**

var myObj = {

top: "hat",

bottom: "pants"

};

myObj.hasOwnProperty("top");

myObj.hasOwnProperty("middle");

The first hasOwnProperty returns true, while the second returns false.

Modify the function checkObj to test if an object passed to the function (obj) contains a specific property (checkProp). If the property is found, return that property's value. If not, return "Not Found".

var Obj = {

gift: "poney",

pet: "kitten",

bed: "sleigh",

city: "Seattle",

}

function checkObj(obj, checkProp) {

if(obj.hasOwnProperty(checkProp)) {

return obj[checkProp];

} else {

return "Not Found";

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**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 JSON is a related data interchange format used to store data.

{

"artist": "Daft Punk",

"title": "Homework",

"release\_year": 1997,

"formats": [

"CD",

"Cassette",

"LP"

],

"gold": true

}

**Note:** You will need to place a comma after every object in the array, unless it is the last object in the array.

Add a new album to the myMusic array. Add artist and title strings, release\_year number, and a formats array of strings.

var myMusic = [

{

"artist": "Billy Joel",

"title": "Piano Man",

"release\_year": 1973,

"formats": [

"CD",

"8T",

"LP"

],

"gold": true,

},

{

"artist": "BSDD",

"title": "KJ",

"release\_year": 1954,

"formats": [

"ADD",

"DSA",

"FDF"

],

"gold": true,

}

];

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Accessing Nested Objects**

The sub-properties of objects can be accessed by chaining together the dot or bracket notation.

Here is a nested object:

var ourStorage = {

"desk": {

"drawer": "stapler"

},

"cabinet": {

"top drawer": {

"folder1": "a file",

"folder2": "secrets"

},

"bottom drawer": "soda"

}

};

ourStorage.cabinet["top drawer"].folder2;

ourStorage.desk.drawer;

ourStorage.cabinet["top drawer"].folder2 would be the string secrets, and ourStorage.desk.drawer would be the string stapler.

Access the myStorage object and assign the contents of the glove box property to the gloveBoxContents variable. Use dot notation for all properties where possible, otherwise use bracket notation.

Run the Tests

Reset All Code

Get Help

var myStorage = {

"car": {

"inside": {

"glove box": "maps",

"passenger seat": "crumbs"

},

"outside": {

"trunk": "jack"

}

}

};

var gloveBoxContents = undefined;

gloveBoxContents=myStorage.car.inside["glove box"];

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Accessing Nested Arrays**

As we have seen in earlier examples, objects can contain both nested objects and nested arrays. Similar to accessing nested objects, array bracket notation can be chained to access nested arrays.

Here is an example of how to access a nested array:

var ourPets = [

{

animalType: "cat",

names: [

"Meowzer",

"Fluffy",

"Kit-Cat"

]

},

{

animalType: "dog",

names: [

"Spot",

"Bowser",

"Frankie"

]

}

];

ourPets[0].names[1];

ourPets[1].names[0];

ourPets[0].names[1] would be the string Fluffy, and ourPets[1].names[0] would be the string Spot.

Using dot and bracket notation, set the variable secondTree to the second item in the trees list from the myPlants object.

var myPlants = [

{

type: "flowers",

list: [

"rose",

"tulip",

"dandelion"

]

},

{

type: "trees",

list: [

"fir",

"pine",

"birch"

]

}

];

var secondTree = "";

secondTree=myPlants[1].list[1];

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Record Collection**

Passed

You are given an object literal representing a part of your musical album collection. Each album has a unique id number as its key and several other properties. Not all albums have complete information.

You start with an updateRecords function that takes an object literal, records, containing the musical album collection, an id, a prop (like artist or tracks), and a value. Complete the function using the rules below to modify the object passed to the function.

* Your function must always return the entire record collection object.
* If prop isn't tracks and value isn't an empty string, update or set that album's prop to value.
* If prop is tracks but the album doesn't have a tracks property, create an empty array and add value to it.
* If prop is tracks and value isn't an empty string, add value to the end of the album's existing tracks array.
* If value is an empty string, delete the given prop property from the album.

**Note:** A copy of the recordCollection object is used for the tests.

// Setup

var recordCollection = {

2548: {

albumTitle: 'Slippery When Wet',

artist: 'Bon Jovi',

tracks: ['Let It Rock', 'You Give Love a Bad Name']

},

2468: {

albumTitle: '1999',

artist: 'Prince',

tracks: ['1999', 'Little Red Corvette']

},

1245: {

artist: 'Robert Palmer',

tracks: []

},

5439: {

albumTitle: 'ABBA Gold'

}

};

// Only change code below this line

function updateRecords(collection, id, prop, value) { // changed here.

if(prop !== 'tracks' && value !== "") {

collection[id][prop] = value;

}

if(prop === 'tracks' && collection[id][prop] === undefined) {

collection[id][prop] = [value];

}

if(prop === 'tracks') {

let track = collection[id][prop];

track.push(value)

}

if(value === "") {

delete collection[id][prop]

}

console.log(collection)

return collection;

}

updateRecords(recordCollection, 5439, 'artist', 'ABBA');

Console output:

{ '1245': { artist: 'Robert Palmer', tracks: [] },

'2468':

{ albumTitle: '1999',

artist: 'Prince',

tracks: [ '1999', 'Little Red Corvette' ] },

'2548':

{ albumTitle: 'Slippery When Wet',

artist: 'Bon Jovi',

tracks: [ 'Let It Rock', 'You Give Love a Bad Name' ] },

'5439': { albumTitle: 'ABBA Gold', artist: 'ABBA' } }

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Iterate with JavaScript While Loops**

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

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++;

}

In the code example above, the while loop will execute 5 times and append the numbers 0 through 4 to ourArray.

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

Add the numbers 5 through 0 (inclusive) in descending order to myArray using a while loop.

// Setup

var myArray = [];

var i=5;

while(i>-1){

myArray.push(i);

i--;

}

// Only change code below this line

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Iterate with JavaScript 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 (a; b; c), where a is the intialization statement, b is the condition statement, and c is the final expression.

The initialization statement is executed one time only before the loop starts. It is typically used to define and setup your loop variable.

The condition statement is evaluated at the beginning of every loop iteration and will continue as long as it evaluates to true. When the condition is false at the start of the iteration, the loop will stop executing. This means if the condition starts as false, your loop will never execute.

The final expression is executed at the end of each loop iteration, prior to the next condition check and is usually used to increment or decrement your loop counter.

In the following example we initialize with i = 0 and iterate while our condition i < 5 is true. We'll increment i by 1 in each loop iteration with i++ as our final expression.

var ourArray = [];

for (var i = 0; i < 5; i++) {

ourArray.push(i);

}

ourArray will now have the value [0,1,2,3,4].

Use a for loop to push the values 1 through 5 onto myArray.

// Setup

var myArray = [];

for(var i=1;i<6;i++){

myArray.push(i);console.log(myArray);

}

// Only change code below this line

// running tests

// tests completed

// console output

[ 1 ]

[ 1, 2 ]

[ 1, 2, 3 ]

[ 1, 2, 3, 4 ]

[ 1, 2, 3, 4, 5 ]

[ 1 ]

[ 1, 2 ]

[ 1, 2, 3 ]

[ 1, 2, 3, 4 ]

[ 1, 2, 3, 4, 5 ]

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Iterate Odd Numbers With a For Loop**

For loops don't have to iterate one at a time. By changing our final-expression, we can count by even numbers.

We'll start at i = 0 and loop while i < 10. We'll increment i by 2 each loop with i += 2.

var ourArray = [];

for (var i = 0; i < 10; i += 2) {

ourArray.push(i);

}

ourArray will now contain [0,2,4,6,8]. Let's change our initialization so we can count by odd numbers.

Push the odd numbers from 1 through 9 to myArray using a for loop.

// Setup

var myArray = [];

for(var i=1;i<10;i+=2){

myArray.push(i);console.log(myArray);

}

// Only change code below this line

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Count Backwards With a For Loop**

A for loop can also count backwards, so long as we can define the right conditions.

In order to decrement by two each iteration, we'll need to change our initialization, condition, and final expression.

We'll start at i = 10 and loop while i > 0. We'll decrement i by 2 each loop with i -= 2.

var ourArray = [];

for (var i = 10; i > 0; i -= 2) {

ourArray.push(i);

}

ourArray will now contain [10,8,6,4,2]. Let's change our initialization and final expression so we can count backwards by twos to create an array of descending odd numbers.

Push the odd numbers from 9 through 1 to myArray using a for loop.

// Setup

var myArray = [];

for (var i = 9; i > 0; i -= 2) {

myArray.push(i);console.log(myArray);

}

// running tests

// tests completed

// console output

[ 9 ]

[ 9, 7 ]

[ 9, 7, 5 ]

[ 9, 7, 5, 3 ]

[ 9, 7, 5, 3, 1 ]

[ 9 ]

[ 9, 7 ]

[ 9, 7, 5 ]

[ 9, 7, 5, 3 ]

[ 9, 7, 5, 3, 1 ]

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**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 indexing, which means the last index of the array is length - 1. Our condition for this loop is i < arr.length, which stops the loop when i is equal to length. In this case the last iteration is i === 4 i.e. when i becomes equal to arr.length - 1 and outputs 6 to the console. Then i increases to 5, and the loop terminates because i < arr.length is false.

Declare and initialize a variable total to 0. Use a for loop to add the value of each element of the myArr array to total.

// Setup

var myArr = [ 2, 3, 4, 5, 6];

var total=0;

for(var i=0;i<myArr.length;i++){

total+=myArr[i];

}

// Only change code below this line

Same way but the solution is more clear now…

// Setup

var myArr = [ 2, 3, 4, 5, 6];

var total=0;

for(var i=0;i<myArr.length;i++){

total+=myArr[i];console.log(total);

}

// Only change code below this line

console.log("final ouput:")

console.log(total);

//Console output

2

5

9

14

20

final ouput:

20

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**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.

Modify function multiplyAll so that it returns the product of all the numbers in the sub-arrays of arr.

function multiplyAll(arr) {

var product = 1;

// Only change code below this line

for (var i = 0; i < arr.length; i++){

for (var j = 0;j < arr[i].length;j++){

product = product \* arr[i][j];

}

}

// Only change code above this line

return product;

}

multiplyAll([[1,2],[3,4],[5,6,7]]);

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Profile Lookup**

We have an array of objects representing different people in our contacts lists.

A lookUpProfile function that takes name and a property (prop) as arguments has been pre-written for you.

The function should check if name is an actual contact's firstName and the given property (prop) is a property of that contact.

If both are true, then return the "value" of that property.

If name does not correspond to any contacts then return the string No such contact.

If prop does not correspond to any valid properties of a contact found to match name then return the string No such property.

Run the Tests

Reset All Code

Get Help

// Setup

var contacts = [

{

"firstName": "Akira",

"lastName": "Laine",

"number": "0543236543",

"likes": ["Pizza", "Coding", "Brownie Points"]

},

{

"firstName": "Harry",

"lastName": "Potter",

"number": "0994372684",

"likes": ["Hogwarts", "Magic", "Hagrid"]

},

{

"firstName": "Sherlock",

"lastName": "Holmes",

"number": "0487345643",

"likes": ["Intriguing Cases", "Violin"]

},

{

"firstName": "Kristian",

"lastName": "Vos",

"number": "unknown",

"likes": ["JavaScript", "Gaming", "Foxes"]

}

];

function lookUpProfile(name, prop){

// Only change code below this line

for(var i=0;i<contacts.length;i++){

if(contacts[i].firstName===name){

if(prop in contacts[i]){

return contacts[i][prop];

}

else{return "No such property"};

}

}

return "No such contact";

// Only change code above this line

}

lookUpProfile("Akira", "likes");

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Generate Random Whole Numbers with JavaScript**

It's great that we can generate random decimal numbers, but it's even more useful if we use it to generate random whole numbers.

1. Use Math.random() to generate a random decimal.
2. Multiply that random decimal by 20.
3. Use another function, Math.floor() to round the number down to its nearest whole number.

Remember that Math.random() can never quite return a 1 and, because we're rounding down, it's impossible to actually get 20. This technique will give us a whole number between 0 and 19.

Putting everything together, this is what our code looks like:

Math.floor(Math.random() \* 20);

We are calling Math.random(), multiplying the result by 20, then passing the value to Math.floor() function to round the value down to the nearest whole number.

Use this technique to generate and return a random whole number between 0 and 9

function randomWholeNum() {

// Only change code below this line

return Math.floor(Math.random()\*10) ;

}

**Generate Random Whole Numbers within a Range**

Instead of generating a random whole number between zero and a given number like we did before, we can generate a random whole 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.

Here's the formula we'll use. Take a moment to read it and try to understand what this code is doing:

Math.floor(Math.random() \* (max - min + 1)) + min

Create a function called randomRange that takes a range myMin and myMax and returns a random whole number that's greater than or equal to myMin, and is less than or equal to myMax, inclusive.

function randomRange(myMin, myMax) {

// Only change code below this line

return Math.floor(Math.random()\*(myMax-myMin+1)+myMin);

// Only change code above this line

}

#### **Code Explanation**

* Math.random() generates our random number between 0 and ≈ 0.9.
* Before multiplying it, it resolves the part between parenthesis (myMax - myMin + 1) because of the grouping operator ( ).
* The result of that multiplication is followed by adding myMin and then “rounded” to the largest integer less than or equal to it (eg: 9.9 would result in 9)

If the values were myMin = 1, myMax= 10, one result could be the following:

1. Math.random() = 0.8244326990411024
2. (myMax - myMin + 1) = 10 - 1 + 1 -> 10
3. a \* b = 8.244326990411024
4. c + myMin = 9.244326990411024
5. Math.floor(9.244326990411024) = 9

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Use the parseInt Function**

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

var a = parseInt("007");

The above function converts the string 007 to the integer 7. If the first character in the string can't be converted into a number, then it returns NaN.

Use parseInt() in the convertToInteger function so it converts the input string str into an integer, and returns it.

function convertToInteger(str) {

return parseInt(str);

}

convertToInteger("56");

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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 parseInt() in the convertToInteger function so it converts a binary number to an integer and returns it.

function convertToInteger(str,radix) {

var radix=2;

return parseInt(str,radix);

}

convertToInteger("10011",2);

convertToInteger("111001",2);

convertToInteger("JamesBond",2);

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**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 a ? b : c, where a is the condition, b is the code to run when the condition returns true, and c is the code to run when the condition returns false.

The following function uses an if/else statement to check a condition:

function findGreater(a, b) {

if(a > b) {

return "a is greater";

}

else {

return "b is greater";

}

}

This can be re-written using the conditional operator:

function findGreater(a, b) {

return a > b ? "a is greater" : "b is greater";

}

Use the conditional operator in the checkEqual function to check if two numbers are equal or not. The function should return either the string Equal or the string Not Equal.

function checkEqual(a, b) {

return a==b ? "Equal" :"Not Equal";

}

checkEqual(1, 2);

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Use Multiple Conditional (Ternary) Operators**

In the previous challenge, you used a single conditional operator. You can also chain them together to check for multiple conditions.

The following function uses if, else if, and else statements to check multiple conditions:

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";

}

It is considered best practice to format multiple conditional operators such that each condition is on a separate line, as shown above. Using multiple conditional operators without proper indentation may make your code hard to read. For example:

function findGreaterOrEqual(a, b) {

return (a === b) ? "a and b are equal" : (a > b) ? "a is greater" : "b is greater";

}

In the checkSign function, use multiple conditional operators - following the recommended format used in findGreaterOrEqual - to check if a number is positive, negative or zero. The function should return positive, negative or zero.

function checkSign(num) {

return (num>0) ? "positive": (num<0) ? "negative" :"zero";

}

checkSign(10);

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Use Recursion to Create a Countdown**

In a [previous challenge](https://www.freecodecamp.org/learn/javascript-algorithms-and-data-structures/basic-javascript/replace-loops-using-recursion), you learned how to use recursion to replace a for loop. Now, let's look at a more complex function that returns an array of consecutive integers starting with 1 through the number passed to the function.

As mentioned in the previous challenge, there will be a base case. The base case tells the recursive function when it no longer needs to call itself. It is a simple case where the return value is already known. There will also be a recursive call which executes the original function with different arguments. If the function is written correctly, eventually the base case will be reached.

For example, say you want to write a recursive function that returns an array containing the numbers 1 through n. This function will need to accept an argument, n, representing the final number. Then it will need to call itself with progressively smaller values of n until it reaches 1. You could write the function as follows:

function countup(n) {

if (n < 1) {

return [];

} else {

const countArray = countup(n - 1);

countArray.push(n);

return countArray;

}

}

console.log(countup(5));

The value [1, 2, 3, 4, 5] will be displayed in the console.

At first, this seems counterintuitive since the value of n *decreases*, but the values in the final array are *increasing*. This happens because the push happens last, after the recursive call has returned. At the point where n is pushed into the array, countup(n - 1) has already been evaluated and returned [1, 2, ..., n - 1].

We have defined a function called countdown with one parameter (n). The function should use recursion to return an array containing the integers n through 1 based on the n parameter. If the function is called with a number less than 1, the function should return an empty array. For example, calling this function with n = 5 should return the array [5, 4, 3, 2, 1]. Your function must use recursion by calling itself and must not use loops of any kind.

// Only change code below this line

function countdown(n, newArr=[]) {

if (n <= 0) {

return newArr;

}

newArr.push(n);

return countdown(n - 1, newArr)

}

console.log(countdown(5));

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.

var camper = 'James';

var camper = 'David';

console.log(camper);

Here the console will display the string David.

As you can see in the code above, the camper variable is originally declared as James and then overridden to be David. 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. If you were to replace var with let in the variable declarations of the code above, the result would be an error.

let camper = 'James';

let camper = 'David';

This error can be seen in the console of your browser. 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:

"use strict";

x = 3.14;

This will display an error that x is not defined.

Update the code so it only uses the let keyword.

let catName;

let quote;

function catTalk() {

"use strict";

catName = "Oliver";

quote = catName + " says Meow!";

}

catTalk();

\*\*\*\*\*\*

**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.

For example:

var numArray = [];

for (var i = 0; i < 3; i++) {

numArray.push(i);

}

console.log(numArray);

console.log(i);

Here the console will display the values [0, 1, 2] and 3.

With the var keyword, i is declared globally. So when i++ is executed, it updates the global variable. This code is similar to the following:

var numArray = [];

var i;

for (i = 0; i < 3; i++) {

numArray.push(i);

}

console.log(numArray);

console.log(i);

Here the console will display the values [0, 1, 2] and 3.

This behavior will cause problems if you were to create a function and store it for later use inside a for loop that uses the i variable. This is because the stored function will always refer to the value of the updated global i variable.

var printNumTwo;

for (var i = 0; i < 3; i++) {

if (i === 2) {

printNumTwo = function() {

return i;

};

}

}

console.log(printNumTwo());

Here the console will display the value 3.

As you can see, printNumTwo() prints 3 and not 2. This is because the value assigned to i was updated and the printNumTwo() returns the global i and not the value i had when the function was created in the for loop. The let keyword does not follow this behavior:

let printNumTwo;

for (let i = 0; i < 3; i++) {

if (i === 2) {

printNumTwo = function() {

return i;

};

}

}

console.log(printNumTwo());

console.log(i);

Here the console will display the value 2, and an error that i is not defined.

i is not defined because it was not declared in the global scope. It is only declared within the for loop statement. printNumTwo() returned the correct value because three different i variables with unique values (0, 1, and 2) were created by the let keyword within the loop statement.

Fix the code so that i declared in the if statement is a separate variable than i declared in the first line of the function. Be certain not to use the var keyword anywhere in your code.

This exercise is designed to illustrate the difference between how var and let keywords assign scope to the declared variable. When programming a function similar to the one used in this exercise, it is often better to use different variable names to avoid confusion.

function checkScope() {

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;

}

Explanation:

let a = 'hello'; // globally scoped

var b = 'world'; // globally scoped

console.log(window.a); // undefined

console.log(window.b); // 'world'

var a = 'hello';

var a = 'world'; // No problem, 'hello' is replaced.

let b = 'hello';

let b = 'world'; // SyntaxError: Identifier 'b' has already been declared

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Declare a Read-Only Variable with the const Keyword**

The keyword let is not the only new way to declare variables. In ES6, you can also declare variables using 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.

const FAV\_PET = "Cats";

FAV\_PET = "Dogs";

The console will display an error due to reassigning the value of FAV\_PET.

As you can see, trying to reassign a variable declared with const will throw an error. You should always name variables you don't want to reassign using the const keyword. This helps when you accidentally attempt to reassign a variable that is meant to stay constant. A common practice when naming constants is to use all uppercase letters, with words separated by an underscore.

**Note:** It is common for developers to use uppercase variable identifiers for immutable values and lowercase or camelCase for mutable values (objects and arrays). In a later challenge you will see an example of a lowercase variable identifier being used for an array.

Change the code so that all variables are declared using let or const. Use let when you want the variable to change, and const when you want the variable to remain constant. Also, rename variables declared with const to conform to common practices, meaning constants should be in all caps.

function printManyTimes(str) {

// Only change code below this line

const SENTENCE = str + " is cool!";

for (let i = 0; i < str.length; i+=2) {

console.log(SENTENCE);

}

// Only change code above this line

}

printManyTimes("freeCodeCamp");

// running tests

// tests completed

// console output

freeCodeCamp is cool!

freeCodeCamp is cool!

freeCodeCamp is cool!

freeCodeCamp is cool!

freeCodeCamp is cool!

freeCodeCamp is cool!

freeCodeCamp is cool!

freeCodeCamp is cool!

freeCodeCamp is cool!

freeCodeCamp is cool!

freeCodeCamp is cool!

freeCodeCamp is cool!

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

var a=1;

function ab(){

let a=2;

console.log(a);

}

ab();

console.log(a);

Console output:

2

1

let a=1;

function ab(){

let a=2;

console.log(a);

}

ab();

console.log(a);

Console output:

2

1

So here var can be updated but let cant be updated same goes with const it remains same for everything that is assigned to it.

var a=1;

var a=3;

function ab(){

let a=2;

console.log(a);

}

ab();

console.log(a);

Console output:

2

3

**Mutate an Array Declared with const**

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.

However, it is important to understand that objects (including arrays and functions) assigned to a variable using const are still mutable. Using the const declaration only prevents reassignment of the variable identifier.

const s = [5, 6, 7];

s = [1, 2, 3];

s[2] = 45;

console.log(s);

s = [1, 2, 3] will result in an error. The console.log will display the value [5, 6, 45].

As you can see, you can mutate the object [5, 6, 7] itself and the variable s will still point to the altered array [5, 6, 45]. Like all arrays, the array elements in s are mutable, but because const was used, you cannot use the variable identifier s to point to a different array using the assignment operator.

An array is declared as const s = [5, 7, 2]. Change the array to [2, 5, 7] using various element assignments.

const s = [5, 7, 2];

function editInPlace() {

// Only change code below this line

s[0]=2;

s[1]=5;

s[2]=7;

// Using s = [2, 5, 7] would be invalid

// Only change code above this line

}

editInPlace();

**Now the const can’t be changed but it can be altered.**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Using function Object.freeze**

**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";

obj.newProp = "Test";

console.log(obj);

The obj.review and obj.newProp assignments will result in errors, and the console will display the value { name: "FreeCodeCamp", review: "Awesome" }.

In this challenge you are going to use Object.freeze to prevent mathematical constants from changing. You need to freeze the MATH\_CONSTANTS object so that no one is able to alter the value of PI, add, or delete properties.

function freezeObj() {

const MATH\_CONSTANTS = {

PI: 3.14

};

// Only change code below this line

Object.freeze(MATH\_CONSTANTS);

// Only change code above this line

try {

MATH\_CONSTANTS.PI = 99;

} catch(ex) {

console.log(ex);

}

return MATH\_CONSTANTS.PI;

}

const PI = freezeObj();

**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;

}

ES6 provides us with the syntactic sugar to not have to write anonymous functions this way. Instead, you can use **arrow function syntax**:

const myFunc = () => {

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 the string value by default.

Rewrite the function assigned to the variable magic which returns a new Date() to use arrow function syntax. Also, make sure nothing is defined using the keyword var.

const magic = () => new Date;

\*\*\*\*\*\*\*\*\*\*\*\*

**Write Arrow Functions with Parameters**

Just like a regular function, you can pass arguments into an arrow function.

const doubler = (item) => item \* 2;

doubler(4);

doubler(4) would return the value 8.

If an arrow function has a single parameter, the parentheses enclosing the parameter may be omitted.

const doubler = item => item \* 2;

It is possible to pass more than one argument into an arrow function.

const multiplier = (item, multi) => item \* multi;

multiplier(4, 2);

multiplier(4, 2) would return the value 8.

Rewrite the myConcat function which appends contents of arr2 to arr1 so that the function uses arrow function syntax.

const myConcat = (arr1, arr2)=> arr1.concat(arr2);

console.log(myConcat([1, 2], [3, 4, 5]));

Console output;

// running tests

// tests completed

// console output

[ 1, 2, 3, 4, 5 ]

[ 1, 2, 3, 4, 5 ]

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**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:

const greeting = (name = "Anonymous") => "Hello " + name;

console.log(greeting("John"));

console.log(greeting());

The console will display the strings Hello John and 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 name will 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.

Modify the function increment by adding default parameters so that it will add 1 to number if value is not specified.

// Only change code below this line

const increment = (number, value=1) => number + value;

// Only change code above this line

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Use the Rest Parameter with Function Parameters**

In order to help us create more flexible functions, ES6 introduces the rest parameter for function parameters. With the rest parameter, 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.

Check out this code:

function howMany(...args) {

return "You have passed " + args.length + " arguments.";

}

console.log(howMany(0, 1, 2));

console.log(howMany("string", null, [1, 2, 3], { }));

The console would display the strings You have passed 3 arguments. and You have passed 4 arguments..

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

Modify the function sum using the rest parameter in such a way that the function sum is able to take any number of arguments and return their sum.

const sum = (...args) => {

return args.reduce((a, b) => a + b, 0);

}

function fun(...input){

let sum = 0;

for(let i of input){

sum+=i;

}

return sum;

}

console.log(fun(1,2)); //3

console.log(fun(1,2,3)); //6

console.log(fun(1,2,3,4,5)); //15

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Use the Spread Operator to Evaluate Arrays In-Place**

ES6 introduces the spread operator, which allows us to expand arrays and other expressions in places where multiple parameters or elements are expected.

The ES5 code below uses apply() to compute the maximum value in an array:

var arr = [6, 89, 3, 45];

var maximus = Math.max.apply(null, arr);

maximus would have a value of 89.

We had to use Math.max.apply(null, arr) because Math.max(arr) returns NaN. Math.max() expects comma-separated arguments, but not an array. The spread operator makes this syntax much better to read and maintain.

const arr = [6, 89, 3, 45];

const maximus = Math.max(...arr);

maximus would have a value of 89.

...arr returns an unpacked array. In other words, it *spreads* the array. However, the spread operator only works in-place, like in an argument to a function or in an array literal. The following code will not work:

const spreaded = ...arr;

Copy all contents of arr1 into another array arr2 using the spread operator.

const arr1 = ['JAN', 'FEB', 'MAR', 'APR', 'MAY'];

let arr2;

arr2 = [...arr1]; // Change this line

console.log(arr2);

// running tests

// tests completed

// console output

[ 'JAN', 'FEB', 'MAR', 'APR', 'MAY' ]

[ 'JAN', 'FEB', 'MAR', 'APR', 'MAY' ]

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Use Destructuring Assignment to Extract Values from Objects**

Destructuring assignment is special syntax introduced in ES6, for neatly assigning values taken directly from an object.

Consider the following ES5 code:

const user = { name: 'John Doe', age: 34 };

const name = user.name;

const age = user.age;

name would have a value of the string John Doe, and age would have the number 34.

Here's an equivalent assignment statement using the ES6 destructuring syntax:

const { name, age } = user;

Again, name would have a value of the string John Doe, and age would have the number 34.

Here, the name and age variables will be created and assigned the values of their respective values from the user object. You can see how much cleaner this is.

You can extract as many or few values from the object as you want.

Replace the two assignments with an equivalent destructuring assignment. It should still assign the variables today and tomorrow the values of today and tomorrow from the HIGH\_TEMPERATURES object.

Run the Tests

Reset All Code

Get Help

const HIGH\_TEMPERATURES = {

yesterday: 75,

today: 77,

tomorrow: 80

};

// Only change code below this line

const {today,tomorrow}=HIGH\_TEMPERATURES;

// Only change code above this line

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Use Destructuring Assignment to Assign Variables from Objects**

Destructuring allows you to assign a new variable name when extracting values. You can do this by putting the new name after a colon when assigning the value.

Using the same object from the last example:

const user = { name: 'John Doe', age: 34 };

Here's how you can give new variable names in the assignment:

const { name: userName, age: userAge } = user;

You may read it as "get the value of user.name and assign it to a new variable named userName" and so on. The value of userName would be the string John Doe, and the value of userAge would be the number 34.

Replace the two assignments with an equivalent destructuring assignment. It should still assign the variables highToday and highTomorrow the values of today and tomorrow from the HIGH\_TEMPERATURES object.

const HIGH\_TEMPERATURES = {

yesterday: 75,

today: 77,

tomorrow: 80

};

// Only change code below this line

const {today:highToday,tomorrow:highTomorrow}=HIGH\_TEMPERATURES;

// Only change code above this line

**Use Destructuring Assignment to Assign Variables from Nested Objects**

You can use the same principles from the previous two lessons to destructure values from nested objects.

Using an object similar to previous examples:

const user = {

johnDoe: {

age: 34,

email: 'johnDoe@freeCodeCamp.com'

}

};

Here's how to extract the values of object properties and assign them to variables with the same name:

const { johnDoe: { age, email }} = user;

And here's how you can assign an object properties' values to variables with different names:

const { johnDoe: { age: userAge, email: userEmail }} = user;

Replace the two assignments with an equivalent destructuring assignment. It should still assign the variables lowToday and highToday the values of today.low and today.high from the LOCAL\_FORECAST object.

const LOCAL\_FORECAST = {

yesterday: { low: 61, high: 75 },

today: { low: 64, high: 77 },

tomorrow: { low: 68, high: 80 }

};

// Only change code below this line

const{today:{low:lowToday,high:highToday}}=LOCAL\_FORECAST;

// Only change code above this line

const LOCAL\_FORECAST = {

yesterday: { low: 61, high: 75 },

today: { low: 64, high: 77 },

tomorrow: { low: 68, high: 80 }

};

// Only change code below this line

const{today:{low:lowToday,high:highToday}}=LOCAL\_FORECAST;

// Only change code above this line

console.log(lowToday);

console.log(highToday);

Console output:

64

77

\*\*\*\*\*\*\*\*\*\*\*\*

**Use 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);

The console will display the values of a and b as 1, 2.

The variable a is assigned the first value of the array, and b is assigned the second value of the array. We can also access the value at any index in an array with destructuring by using commas to reach the desired index:

const [a, b,,, c] = [1, 2, 3, 4, 5, 6];

console.log(a, b, c);

The console will display the values of a, b, and c as 1, 2, 5.

Use destructuring assignment to swap the values of a and b so that a receives the value stored in b, and b receives the value stored in a.

let a = 8, b = 6;

// Only change code below this line

[a,b]=[b,a];

console.log(a);

console.log(b);

6

8

\*\*\*\*\*\*\*

**Use Destructuring Assignment with the Rest Parameter 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);

console.log(arr);

The console would display the values 1, 2 and [3, 4, 5, 7].

Variables a and b take the first and second values from the array. After that, because of the rest parameter's presence, arr gets the rest of the values in the form of an array. The rest element only works correctly as the last variable in the list. As in, you cannot use the rest parameter to catch a subarray that leaves out the last element of the original array.

Use destructuring assignment with the rest parameter to perform an effective Array.prototype.slice() so that arr is a sub-array of the original array source with the first two elements omitted.

const source = [1,2,3,4,5,6,7,8,9,10];

function removeFirstTwo(list) {

// Only change code below this line

const [a,b,...arr] = list; // Change this line

// Only change code above this line

return arr;

}

const arr = removeFirstTwo(source);

\*\*\*\*\*\*\*\*\*\*\*\*\*

**Use Destructuring Assignment to Pass an Object as a Function's Parameters**

In some cases, you can destructure the object in a function argument itself.

Consider the code below:

const profileUpdate = (profileData) => {

const { name, age, nationality, location } = profileData;

}

This effectively destructures the object sent into the function. This can also be done in-place:

const profileUpdate = ({ name, age, nationality, location }) => {

}

When profileData is passed to the above function, the values are destructured from the function parameter for use within the function.

Use destructuring assignment within the argument to the function half to send only max and min inside the function.

const stats = {

max: 56.78,

standard\_deviation: 4.34,

median: 34.54,

mode: 23.87,

min: -0.75,

average: 35.85

};

// Only change code below this line

const half = ({min,max}) => (max + min) / 2.0;

// Only change code above this line

\*\*\*\*\*\*\*\*\*\*\*\*

**Create Strings using Template Literals**

A new feature of ES6 is the template literal. This is a special type of string that makes creating complex strings easier.

Template literals allow you to create multi-line strings and to use string interpolation features to create strings.

Consider the code below:

const person = {

name: "Zodiac Hasbro",

age: 56

};

const greeting = `Hello, my name is ${person.name}!

I am ${person.age} years old.`;

console.log(greeting);

The console will display the strings Hello, my name is Zodiac Hasbro! and I am 56 years old..

A lot of things happened there. Firstly, the example uses backticks (`), not quotes (' or "), to wrap the string. Secondly, notice that the string is multi-line, both in the code and the output. This saves inserting \n within strings. The ${variable} syntax used above is a placeholder. Basically, you won't have to use concatenation with the + operator anymore. To add variables to strings, you just drop the variable in a template string and wrap it with ${ and }. Similarly, you can include other expressions in your string literal, for example ${a + b}. This new way of creating strings gives you more flexibility to create robust strings.

Use template literal syntax with backticks to create an array of list element (li) strings. Each list element's text should be one of the array elements from the failure property on the result object and have a class attribute with the value text-warning. The makeList function should return the array of list item strings.

Use an iterator method (any kind of loop) to get the desired output (shown below).

[

'<li class="text-warning">no-var</li>',

'<li class="text-warning">var-on-top</li>',

'<li class="text-warning">linebreak</li>'

]

const result = {

success: ["max-length", "no-amd", "prefer-arrow-functions"],

failure: ["no-var", "var-on-top", "linebreak"],

skipped: ["no-extra-semi", "no-dup-keys"]

};

function makeList(arr) {

// Only change code below this line

const failureItems = arr.map((a) => {

return `<li class="text-warning">${a}</li>`

});

// Only change code above this line

return failureItems;

}

const failuresList = makeList(result.failure);

\*\*\*\*\*\*

**Write Concise Object Literal Declarations Using Object Property Shorthand**

ES6 adds some nice support for easily defining object literals.

Consider the following code:

const getMousePosition = (x, y) => ({

x: x,

y: y

});

getMousePosition is a simple function that returns an object containing two properties. ES6 provides the syntactic sugar to eliminate the redundancy of having to write x: x. You can simply write x once, and it will be converted tox: x (or something equivalent) under the hood. Here is the same function from above rewritten to use this new syntax:

const getMousePosition = (x, y) => ({ x, y });

Use object property shorthand with object literals to create and return an object with name, age and gender properties.

Run the Tests

Reset All Code

Get Help

const createPerson = (name, age, gender) => ({name,age,gender});

// Only change code below this line {

// Only change code above this line

\*\*\*\*\*\*\*\*\*\*

**Write Concise Declarative Functions with ES6**

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

const person = {

name: "Taylor",

sayHello: function() {

return `Hello! My name is ${this.name}.`;

}

};

With ES6, you can remove the function keyword 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}.`;

}

};

Refactor the function setGear inside the object bicycle to use the shorthand syntax described above.

// Only change code below this line

const bicycle = {

gear: 2,

setGear(newGear) {

this.gear = newGear;

}

};

// Only change code above this line

bicycle.setGear(3);

console.log(bicycle.gear);

// running tests

// tests completed

// console output

3

3

Here this.gear is pointing to object bicycle and then going in gear,where value is 2 now in the function setGear=newGear it simply assigns value to the new parameter which is declared as 3=newGear and then gear in bicycle is set to 3 from 2.

\*\*\*\*\*\*\*\*\*\*\*\*\*

**Use class Syntax to Define a Constructor Function**

ES6 provides a new syntax to create objects, using the class keyword.

It should be noted that the class syntax is just syntax, and not a full-fledged class-based implementation of an object-oriented paradigm, unlike in languages such as Java, Python, Ruby, etc.

In ES5, we usually define a constructor function and use the new keyword to instantiate an object.

var SpaceShuttle = function(targetPlanet){

this.targetPlanet = targetPlanet;

}

var zeus = new SpaceShuttle('Jupiter');

The class syntax simply replaces the constructor function creation:

class SpaceShuttle {

constructor(targetPlanet) {

this.targetPlanet = targetPlanet;

}

}

const zeus = new SpaceShuttle('Jupiter');

It should be noted that the class keyword declares a new function, to which a constructor is added. This constructor is invoked when new is called to create a new object.

**Note:** UpperCamelCase should be used by convention for ES6 class names, as in SpaceShuttle used above.

The constructor method is a special method for creating and initializing an object created with a class. You will learn more about it in the Object Oriented Programming section of the JavaScript Algorithms And Data Structures Certification.

Use the class keyword and write a constructor to create the Vegetable class.

The Vegetable class allows you to create a vegetable object with a property name that gets passed to the constructor.

// Only change code below this line

class Vegetable{

constructor(name){

this.name=name;

}

}

// Only change code above this line

const carrot = new Vegetable('carrot');

console.log(carrot.name); // Should display 'carrot'

// running tests

// tests completed

// console output

carrot

carrot

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*8

**Use getters and setters to Control Access to an Object**

You can obtain values from an object and set the value of a property within an object.

These are classically called getters and setters.

Getter functions are meant to simply return (get) the value of an object's private variable to the user without the user directly accessing the private variable.

Setter functions are meant to modify (set) the value of an object's private variable based on the value passed into the setter function. This change could involve calculations, or even overwriting the previous value completely.

class Book {

constructor(author) {

this.\_author = author;

}

// getter

get writer() {

return this.\_author;

}

// setter

set writer(updatedAuthor) {

this.\_author = updatedAuthor;

}

}

const novel = new Book('anonymous');

console.log(novel.writer);

novel.writer = 'newAuthor';

console.log(novel.writer);

The console would display the strings anonymous and newAuthor.

Notice the syntax used to invoke the getter and setter. They do not even look like functions. Getters and setters are important because they hide internal implementation details.

**Note:** It is convention to precede the name of a private variable with an underscore (\_). However, the practice itself does not make a variable private.

Use the class keyword to create a Thermostat class. The constructor accepts a Fahrenheit temperature.

In the class, create a getter to obtain the temperature in Celsius and a setter to set the temperature in Celsius.

Remember that C = 5/9 \* (F - 32) and F = C \* 9.0 / 5 + 32, where F is the value of temperature in Fahrenheit, and C is the value of the same temperature in Celsius.

**Note:** When you implement this, you will track the temperature inside the class in one scale, either Fahrenheit or Celsius.

This is the power of a getter and a setter. You are creating an API for another user, who can get the correct result regardless of which one you track.

In other words, you are abstracting implementation details from the user.

function makeClass() {

"use strict";

/\* Alter code below this line \*/

class Thermostat {

constructor (F) {

this.celsius = 5/9 \* (F - 32);

}

get temperature() {

console.log(`${this.celsius} in C`);

return this.celsius;

}

set temperature(newTemp) {

this.celsius = newTemp;

}

}

/\* Alter code above this line \*/

return Thermostat;

}

const Thermostat = makeClass();

const thermos = new Thermostat(76); // setting in Fahrenheit scale

let temp = thermos.temperature; // 24.44 in C

thermos.temperature = 26;

temp = thermos.temperature; // 26 in C

// running tests

24.444444444444446 in C

26 in C

// tests completed

// console output

24.444444444444446 in C

26 in C

50 in C

26 in C

50 in C

**Create a Module Script**

JavaScript started with a small role to play on an otherwise mostly HTML web. Today, it’s huge, and some websites are built almost entirely with JavaScript. In order to make JavaScript more modular, clean, and maintainable; ES6 introduced a way to easily share code among JavaScript files. This involves exporting parts of a file for use in one or more other files, and importing the parts you need, where you need them. In order to take advantage of this functionality, you need to create a script in your HTML document with a type of module. Here’s an example:

<script type="module" src="filename.js"></script>

A script that uses this module type can now use the import and export features you will learn about in the upcoming challenges.

Add a script to the HTML document of type module and give it the source file of index.js

<html>

<body>

<!-- Only change code below this line -->

<script type="module" src="index.js"></script>

<!-- Only change code above this line -->

</body>

</html>

**Use export to Share a Code Block**

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 };

There are two string-related functions in the editor. Export both of them using the method of your choice.

const uppercaseString = (string) => {

return string.toUpperCase();

}

const lowercaseString = (string) => {

return string.toLowerCase()

}

export{uppercaseString,lowercaseString}

**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';

Add the appropriate import statement that will allow the current file to use the uppercaseString and lowercaseString functions you exported in the previous lesson. These functions are in a file called string\_functions.js, which is in the same directory as the current file.

import{uppercaseString,lowercaseString} from './string\_functions.js';

// Only change code above this line

uppercaseString("hello");

lowercaseString("WORLD!");

**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);

The code in this file requires the contents of the file: string\_functions.js, that is in the same directory as the current file. Use the import \* as syntax to import everything from the file into an object called stringFunctions.

import \* as stringFunctions from "./string\_functions.js";

// Only change code above this line

stringFunctions.uppercaseString("hello");

stringFunctions.lowercaseString("WORLD!");

Now importing everything with star(\*) operator means you have to create an object named stringFunction and all data will be stored in that function.Later you have to call that object.function name and then pass parameter.

**Create an Export Fallback with export default**

In the export lesson, 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 export syntax 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.

Below are examples using export default:

export default function add(x, y) {

return x + y;

}

export default function(x, y) {

return x + y;

}

The first is a named function, and the second is an anonymous function.

Since export default is 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

The following function should be the fallback value for the module. Please add the necessary code to do so.

export default function subtract(x, y) {

return x - y;

}

Explanation:

**Default Exports:** Default exports are useful to export only a single object, function, variable. During the import, we can use any name to import.

|  |
| --- |
| // file module.js  var x=4;  export default x;    // test.js  // while importing x in test.js  import y from './module';  // note that y is used import x instead of  // import x, because x was default export  console.log(y);  // output will be 4 |

**Output**

4

Now remember that export default can be declared only once in the file,while calling export default in other file we can use any name to call it since it is unique to that file we are calling like

eg.Export default function add(a+b){return a+b;} //filename 1

In other file(named as 2 ) calling this function add

Import y from “./1”; // you can call it by any name as every file has different export default

add(2,4);

console.log(add);

Output will be 6.

**Import a Default Export**

In the last challenge, you learned about export default and its uses. To import a default export, you need to use a different import syntax. In the following example, add is the default export of the math\_functions.js file. Here is how to import it:

import add from "./math\_functions.js";

The syntax differs in one key place. The imported value, add, is not surrounded by curly braces ({}). add here is simply a variable name for whatever the default export of the math\_functions.js file is. You can use any name here when importing a default.

In the following code, import the default export from the math\_functions.js file, found in the same directory as this file. Give the import the name subtract.

import subtract from "./math\_functions.js"

// Only change code above this line

subtract(7,4);

**Create a JavaScript Promise**

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) => {

});

Create a new promise called makeServerRequest. Pass in a function with resolve and reject parameters to the constructor.

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

// responseFromServer represents a response from a server

let responseFromServer=a;

if(responseFromServer===a) {

// change this line

resolve("We got the Data");

} else {

// change this line

reject("Data not Received");

}

});

**Complete a Promise with resolve and reject**

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.

Make the promise handle success and failure. If responseFromServer is true, call the resolve method to successfully complete the promise. Pass resolve a string with the value We got the data. If responseFromServer is false, use the reject method instead and pass it the string: Data not received.

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

// responseFromServer represents a response from a server

let responseFromServer;

if(responseFromServer) {

resolve("We got the data");

} else {

reject("Data not received");

}

});

**Handle a Fulfilled Promise with then**

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 => {

});

result comes from the argument given to the resolve method.

Add the then method to your promise. Use result as

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

// responseFromServer is set to true to represent a successful response from a server

let responseFromServer = true;

if(responseFromServer) {

resolve("We got the data");

} else {

reject("Data not received");

}

});

makeServerRequest.then(result =>{

console.log(result);

});

\*\*\*

**Handle a Rejected Promise with catch**

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 => {

});

error is the argument passed in to the reject method.

Add the catch method to your promise. Use error as the parameter of its callback function and log error to the console.

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

// responseFromServer is set to false to represent an unsuccessful response from a server

let responseFromServer = false;

if(responseFromServer) {

resolve("We got the data");

} else {

reject("Data not received");

}

});

makeServerRequest.then(result => {

console.log(result);

});

makeServerRequest.catch(error=>{

console.log(error);

});

\*\*\*\*\*\*\*\*\*\*\*\*

**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);

The test method here returns true.

Apply the regex myRegex on the string myString using the .test() method.

let myString = "Hello, World!";

let myRegex = /Hello/;

let result = myRegex.test(myString); // Change this line

console.log(result);

Console output:

true

\*\*\*\*\*\*\*\*

**Match Literal Strings**

In the last challenge, you searched for the word Hello using the regular expression /Hello/. That regex searched for a literal match of the string Hello. Here's another example searching for a literal match of the string Kevin:

let testStr = "Hello, my name is Kevin.";

let testRegex = /Kevin/;

testRegex.test(testStr);

This test call will return true.

Any other forms of Kevin will not match. For example, the regex /Kevin/ will not match kevin or KEVIN.

let wrongRegex = /kevin/;

wrongRegex.test(testStr);

This test call will return false.

A future challenge will show how to match those other forms as well.

Complete the regex waldoRegex to find "Waldo" in the string waldoIsHiding with a literal match.

let waldoIsHiding = "Somewhere Waldo is hiding in this text.";

let waldoRegex = /Waldo/; // Change this line

let result = waldoRegex.test(waldoIsHiding);

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**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 the strings 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/.

Complete the regex petRegex to match the pets dog, cat, bird, or fish.

var petString = "James has a pet cat.";

var petRegex = /dog|cat|bird|fish/; // Change this line

var result = petRegex.test(petString);

\*\*\*\*\*

**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.

Write a regex fccRegex to match freeCodeCamp, no matter its case. Your regex should not match any abbreviations or variations with spaces.

let myString = "freeCodeCamp";

let fccRegex = /freecodecamp/i; // Change this line

let result = fccRegex.test(myString);

\*\*\*\*\*

**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/);

let ourStr = "Regular expressions";

let ourRegex = /expressions/;

ourStr.match(ourRegex);

Here the first match would return ["Hello"] and the second would return ["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');

Apply the .match() method to extract the string coding.

let extractStr = "Extract the word 'coding' from this string.";

let codingRegex = /coding/; // Change this line

let result = extractStr.match(codingRegex); // Change this line

console.log(result);

Console output:

[ 'coding',

index: 18,

input: 'Extract the word \'coding\' from this string.',

groups: undefined ]

.match() extracts the entire output that is given in parenthesess like it was given coding so it actually return coding when found rather than returning true or false in test case which we saw in previous examples.

\*\*\*\*\*\*\*\*\*\*\*\*\*

**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);

Here match would return ["Repeat"].

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

let repeatRegex = /Repeat/g;

testStr.match(repeatRegex);

And here match returns the value ["Repeat", "Repeat", "Repeat"]

Using the regex starRegex, find and extract both Twinkle words from the string twinkleStar.

**Note**

You can have multiple flags on your regex like /search/gi

let twinkleStar = "Twinkle, twinkle, little star";

let starRegex = /Twinkle/ig; // Change this line

let result = twinkleStar.match(starRegex); // Change this line

\*\*\*\*\*\*\*\*\*\*\*\*

**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);

huRegex.test(hugStr);

Both of these test calls would return true.

Complete the regex unRegex so that it matches the strings run, sun, fun, pun, nun, and bun. Your regex should use the wildcard character.

let exampleStr = "Let's have fun with regular expressions!";

let unRegex = /un.|run|sun|fun|pun|nun|bun/; // Change this line

let result = unRegex.test(exampleStr);

\*\*\*\*\*\*\*\*\*\*\*\*

**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);

bagStr.match(bgRegex);

bugStr.match(bgRegex);

bogStr.match(bgRegex);

In order, the four match calls would return the values ["big"], ["bag"], ["bug"], and null.

Use a character class with vowels (a, e, i, o, u) in your regex vowelRegex to find all the vowels in the string quoteSample.

**Note:** Be sure to match both upper- and lowercase vowels.

let quoteSample = "Beware of bugs in the above code; I have only proved it correct, not tried it.";

let vowelRegex = /[aeiou]/gi; // Change this line

let result =quoteSample.match(vowelRegex); // Change this line

console.log(result);

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**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);

batStr.match(bgRegex);

matStr.match(bgRegex);

In order, the three match calls would return the values ["cat"], ["bat"], and null.

Match all the letters in the string quoteSample.

**Note**: Be sure to match both uppercase and lowercase letters.

let quoteSample = "The quick brown fox jumps over the lazy dog.";

let alphabetRegex = /[a-z]/ig; // Change this line

let result = quoteSample.match(alphabetRegex); // Change this line

\*\*\*\*\*\*\*

**Match Numbers and Letters of the Alphabet**

Using the hyphen (-) to match a range of characters is not limited to letters. It also works to match a range of numbers.

For example, /[0-5]/ matches any number between 0 and 5, including the 0 and 5.

Also, it is possible to combine a range of letters and numbers in a single character set.

let jennyStr = "Jenny8675309";

let myRegex = /[a-z0-9]/ig;

jennyStr.match(myRegex);

Create a single regex that matches a range of letters between h and s, and a range of numbers between 2 and 6. Remember to include the appropriate flags in the regex.

let quoteSample = "Blueberry 3.141592653s are delicious.";

let myRegex = /[h-s2-6]/gi; // Change this line

let result = quoteSample.match(myRegex); // Change this line

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**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.

Create a single regex that matches all characters that are not a number or a vowel. Remember to include the appropriate flags in the regex.

let quoteSample = "3 blind mice.";

let myRegex = /[^aeiou0-9]/ig; // Change this line

let result =quoteSample.match( myRegex); // Change this line

console.log(result);

\*\*\*\*\*\*\*\*\*\*\*

**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.

You want to find matches when the letter s occurs one or more times in Mississippi. Write a regex that uses the + sign.

let difficultSpelling = "Mississippi";

let myRegex = /s+/g; // Change this line

let result = difficultSpelling.match(myRegex);

\*\*\*\*\*\*\*\*\*\*\*\*

**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);

gPhrase.match(goRegex);

oPhrase.match(goRegex);

In order, the three match calls would return the values ["goooooooo"], ["g"], and null.

For this challenge, chewieQuote has been initialized as the string Aaaaaaaaaaaaaaaarrrgh! behind the scenes. Create a regex chewieRegex that uses the \* character to match an uppercase A character immediately followed by zero or more lowercase a characters in chewieQuote. Your regex does not need flags or character classes, and it should not match any of the other quotes.

// Only change code below this line

let chewieRegex = /Aa\*/; // Change this line

// Only change code above this line

let result = chewieQuote.match(chewieRegex);

console.log(result);

\*\*\*\*\*\*\*\*\*\*

**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"].

**Note:** Parsing HTML with regular expressions should be avoided, but pattern matching an HTML string with regular expressions is completely fine.

Fix the regex /<.\*>/ to return the HTML tag <h1> and not the text "<h1>Winter is coming</h1>". Remember the wildcard . in a regular expression matches any character.

let text = "<h1>Winter is coming</h1>";

let myRegex = /<.\*?>/; // Change this line

let result = text.match(myRegex);

console.log(result);

\*\*\*\*\*\*\*\*

**Find One or More Criminals in a Hunt**

Time to pause and test your new regex writing skills. A group of criminals escaped from jail and ran away, but you don't know how many. However, you do know that they stay close together when they are around other people. You are responsible for finding all of the criminals at once.

Here's an example to review how to do this:

The regex /z+/ matches the letter z when it appears one or more times in a row. It would find matches in all of the following strings:

"z"

"zzzzzz"

"ABCzzzz"

"zzzzABC"

"abczzzzzzzzzzzzzzzzzzzzzabc"

But it does not find matches in the following strings since there are no letter z characters:

""

"ABC"

"abcabc"

Write a greedy regex that finds one or more criminals within a group of other people. A criminal is represented by the capital letter C.

let reCriminals = /C+/; // Change this line

\*\*\*\*\*\*\*\*\*\*\*\*

**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 caret character (^) inside a character set to create a negated character set in the form [^thingsThatWillNotBeMatched]. Outside of a character set, the caret is 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);

let notFirst = "You can't find Ricky now.";

firstRegex.test(notFirst);

The first test call would return true, while the second would return false.

Use the caret character in a regex to find Cal only in the beginning of the string rickyAndCal.

let rickyAndCal = "Cal and Ricky both like racing.";

let calRegex = /^Cal/; // Change this line

let result = calRegex.test(rickyAndCal);

\*\*\*\*\*\*\*\*\*\*

**Match Ending String Patterns**

In the last challenge, you learned to use the caret character to search for patterns at the beginning of strings. There is also a way to search for patterns at the end of strings.

You can search the end of strings using the dollar sign character $ at the end of the regex.

let theEnding = "This is a never ending story";

let storyRegex = /story$/;

storyRegex.test(theEnding);

let noEnding = "Sometimes a story will have to end";

storyRegex.test(noEnding);

The first test call would return true, while the second would return false.

Use the anchor character ($) to match the string caboose at the end of the string caboose.

let caboose = "The last car on a train is the caboose";

let lastRegex = /caboose$/; // Change this line

let result = lastRegex.test(caboose);

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Match All Letters and Numbers**

Using character classes, you were able to search for all letters of the alphabet with [a-z]. This kind of character class is common enough that there is a shortcut for it, although it includes a few extra characters as well.

The closest character class in JavaScript to match the alphabet is \w. This shortcut is equal to [A-Za-z0-9\_]. This character class matches upper and lowercase letters plus numbers. Note, this character class also includes the underscore character (\_).

let longHand = /[A-Za-z0-9\_]+/;

let shortHand = /\w+/;

let numbers = "42";

let varNames = "important\_var";

longHand.test(numbers);

shortHand.test(numbers);

longHand.test(varNames);

shortHand.test(varNames);

All four of these test calls would return true.

These shortcut character classes are also known as shorthand character classes.

Use the shorthand character class \w to count the number of alphanumeric characters in various quotes and strings.

let quoteSample = "The five boxing wizards jump quickly.";

let alphabetRegexV2 = /\w/g; // Change this line

let result = quoteSample.match(alphabetRegexV2).length;

\*\*\*\*\*\*

**Match Everything But Letters and Numbers**

You've learned that you can use a shortcut to match alphanumerics [A-Za-z0-9\_] using \w. A natural pattern you might want to search for is the opposite of alphanumerics.

You can search for the opposite of the \w with \W. Note, the opposite pattern uses a capital letter. This shortcut is the same as [^A-Za-z0-9\_].

let shortHand = /\W/;

let numbers = "42%";

let sentence = "Coding!";

numbers.match(shortHand);

sentence.match(shortHand);

The first match call would return the value ["%"] and the second would return ["!"].

Use the shorthand character class \W to count the number of non-alphanumeric characters in various quotes and strings.

let quoteSample = "The five boxing wizards jump quickly.";

let nonAlphabetRegex = /\W/g; // Change this line

let result = quoteSample.match(nonAlphabetRegex).length;

\w is [A-Za-z0-9\_]

Means \W is opposite of \w which is [^A-Za-z0-9\_]

**Match All Numbers**

You've learned shortcuts for common string patterns like alphanumerics. Another common pattern is looking for just digits or numbers.

The shortcut to look for digit characters is \d, with a lowercase d. This is equal to the character class [0-9], which looks for a single character of any number between zero and nine.

Use the shorthand character class \d to count how many digits are in movie titles. Written out numbers ("six" instead of 6) do not count.

let movieName = "2001: A Space Odyssey";

let numRegex = /\d/g; // Change this line

let result = movieName.match(numRegex).length;

console.log(result);

\*\*\*\*\*

**Match All Non-Numbers**

The last challenge showed how to search for digits using the shortcut \d with a lowercase d. You can also search for non-digits using a similar shortcut that uses an uppercase D instead.

The shortcut to look for non-digit characters is \D. This is equal to the character class [^0-9], which looks for a single character that is not a number between zero and nine.

Use the shorthand character class for non-digits \D to count how many non-digits are in movie titles.

let movieName = "2001: A Space Odyssey";

let noNumRegex = /\D/g; // Change this line

let result = movieName.match(noNumRegex).length;

Now \d is[0-9] but \D is [^0-9].

\*\*\*\*\*\*\*\*\*\*\*\*\*

**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.

Change the regex userCheck to fit the constraints listed above.

let username = "JackOfAllTrades";

let userCheck = /^[a-z]([0-9]{2,}|[a-z]+\d\*)$/i; // Change this line

let result = userCheck.test(username);

Explanation:

let username = “JackOfAllTrades”;

let userCheck = /change/; // Change this line

let result = userCheck.test(username);

Tests that must pass:

Your regex should match JACK

Your regex should not match J

Your regex should match Jo

Your regex should match Oceans11

Your regex should match RegexGuru

Your regex should not match 007

Your regex should not match 9

Your regex should not match A1

Your regex should not match BadUs3rnam3

Your regex should match Z97

Your regex should not match c57bT3

#### **Answer Breakdown**

Let’s start building out the features that are required one-by-one

**Requirement 1: usernames can only have alphanumeric characters.**

That’s easy, it should be [a-z] or [0-9]

**Requirement 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.**

So they are basically saying TWO things:

Letters HAVE to be at the beginning. You have to have at least 1 letter, at the beginning

Numbers are optional, but they HAVE to be at the end.

The carat (^) and dollar sign ($) can support these two reqiurements. ^ is the start of the input, while $ is the end of the input.

This is the part that confused me:

In order to end with 0 or more numbers, we need to do this:

\d\*

The asterisk (\*) means “zero or more”. Maybe one way to remember this as a “star” as being a “point” in the sky (but don’t get it confused with the period, which I believe is a wildcard for a single character). A “point” is something that literally has zero dimensions. A flat line has 1 dimension because it has a length. But a constellation has 2 dimensions (if you draw straight lines from them), or something with 3 dimensions (if you remember that the stars aren’t the same distance away from you), or even more dimensions (if you believe the black hole/white hole/ alternate universe conspiracies). Yeah, I’m definitely not going to forget what an asterisk means after coming up with this example.

**3. usernames can be uppercase or lowercase. We just need an “i” after the second forward slash (/) for this. Better than doing [a-zA-Z]**

**4. Usernames should be at least two cahracters long. So this gives us two options:**

One letter + any number of any other characters. But the first character HAS to be a letter, according to rule number 1.

So let’s start piecing this together:

Step 1: /[a-z0-9]/

Everything has to be alphanumeric

Step 2: Letters have to be at the beginning, and numbers have to be at the end.

The first character HAS to be a letter, that’s why we keep that initial [a-z] separately

/^[a-z][0-9]$\d\*/

Step 3: Usernames can be uppercase or lowercase

/^[a-z][0-9]\d\*$/i <–that i means it will match lowercase or uppercase

Step 4: At least 2 characters long

let username = “JackOfAllTrades”;

const userCheck = /^[a-z]([0-9]{2,}|[a-z]+\d\*)$/i;

let result = userCheck.test(username);

**Let’s break down what this means:**

const userCheck = **/^[a-z]([0-9]{2,}|[a-z]+\d\*)$/i**;

/ /

The two forward slashes at the beginning and the end just let us know, hey, this is ES6.

**^**

The carat means that we HAVE to start with [a-z]

**[a-z]**

And what does [a-z] mean? It means it includes any letter in the alphabet, though it only covers lowercase letters.

**i**

That’s why, after the second forward slash (/), we need an “i” to show that we want all letters from [a-z] AND all capital letters too, from [A-Z] as well. The “i” means, we want both lowercase and capital letters

**()**

The parentheses are used for grouping things together

**|**

this just means “or.” So we can either have one of these two options after [a-z]:

[0-9]{2,} OR [a-z]+\d\*

**[0-9]**

means a number between 0 through 9

**{2,}**

Rexegg.com, which is an awesome (albeit somewhat outdated when it comes to ES6) resource that I found online, explains it very well on their quick reference table. Whenever you have curly brackets {}, the first number will represent your starting point, or the minimum number of times, and the second number, after the comma, represents your ending point, or the maximum number of times.

So it’s {min, max}

If it’s like this, then its min times, or more {min,}

If it’s like this, then it’s min to max times {min, max}

If it’s like this, then it’s exactly number times {number}

So {2,}, because it does NOT have a second number BUT it DOES have a comma, means {min,}, or between 2 times and any other number of times.

After the OR sign (|) ,we have this:

**[a-z]**

Once again, letters a to z, lowercase, and inclusive [although the “i” at the very end makes it include A-Z letters as well]

**+**

The plus sign means 1 or more. You can probably also write it as {1,}

Another way that we add one or more, if you remember from for loops, is to write this:

Instead of i += 1, we do i++

That means to add 1 (or to +1)

Here, + in regex means 1 or more (or 1+)

Just think of switching around that plus sign to the back (1+) instead of at the front (+1)

**\d**

This means any number, or 0-9

**\***

An asterisk or star is a single point in the sky. A point has ZERO dimensions as a point. But a constellation is 2D if you draw it out with your finger (a flat shape like the trapezoid that makes up the scoop of the Big dipper is two-dimensional). And 3D if you remember that stars in a constellation are different distances away from each other and from you. And 4D if you believe that stars are gateways to other dimensions.

So I remember the asterisk or star as meaning “0+”

Let’s translate this into a sentence:

/^[a-z]([0-9]{2,}|[a-z]+\d\*$/i

“We’re going to do something cool with regex (/) now.

So let’s look for something that starts with (^) the letters [a-z].

And let’s look for something that ends with the number [0-9], at least {2, or more} digits

OR

something that starts with the letters [a-z],

and we can end it with ($) one or more (+) letters, and as many (\*) numbers (\d) as we would like.

Let’s make sure that we get both lowercase and UPPERCASE letters (i)”

\*\*\*\*\*

**Match Whitespace**

The challenges so far have covered matching letters of the alphabet and numbers. You can also match the whitespace or spaces between letters.

You can search for whitespace using \s, which is a lowercase s. This pattern not only matches whitespace, but also carriage return, tab, form feed, and new line characters. You can think of it as similar to the character class [ \r\t\f\n\v].

let whiteSpace = "Whitespace. Whitespace everywhere!"

let spaceRegex = /\s/g;

whiteSpace.match(spaceRegex);

This match call would return [" ", " "].

Change the regex countWhiteSpace to look for multiple whitespace characters in a string.

let sample = "Whitespace is important in separating words";

let countWhiteSpace = /\s/g; // Change this line

let result = sample.match(countWhiteSpace);

\*\*\*\*

**Match Non-Whitespace Characters**

You learned about searching for whitespace using \s, with a lowercase s. You can also search for everything except whitespace.

Search for non-whitespace using \S, which is an uppercase s. This pattern will not match whitespace, carriage return, tab, form feed, and new line characters. You can think of it being similar to the character class [^ \r\t\f\n\v].

let whiteSpace = "Whitespace. Whitespace everywhere!"

let nonSpaceRegex = /\S/g;

whiteSpace.match(nonSpaceRegex).length;

The value returned by the .length method would be 32.

Change the regex countNonWhiteSpace to look for multiple non-whitespace characters in a string.

let sample = "Whitespace is important in separating words";

let countNonWhiteSpace = /\S/g; // Change this line

let result = sample.match(countNonWhiteSpace);

\*\*\*\*\*\*\*\*

**Specify Upper and Lower Number of Matches**

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.

For example, to match only the letter a appearing between 3 and 5 times in the string ah, your regex would be /a{3,5}h/.

let A4 = "aaaah";

let A2 = "aah";

let multipleA = /a{3,5}h/;

multipleA.test(A4);

multipleA.test(A2);

The first test call would return true, while the second would return false.

Change the regex ohRegex to match the entire phrase Oh no only when it has 3 to 6 letter h's.

let ohStr = "Ohhh no";

let ohRegex = /oh{3,6}\sno/i; // Change this line

let result = ohRegex.test(ohStr);

\*\*

**Specify Only the Lower Number of Matches**

You can specify the lower and upper number of patterns with quantity specifiers using curly brackets. Sometimes you only want to specify the lower number of patterns with no upper limit.

To only specify the lower number of patterns, keep the first number followed by a comma.

For example, to match only the string hah with the letter a appearing at least 3 times, your regex would be /ha{3,}h/.

let A4 = "haaaah";

let A2 = "haah";

let A100 = "h" + "a".repeat(100) + "h";

let multipleA = /ha{3,}h/;

multipleA.test(A4);

multipleA.test(A2);

multipleA.test(A100);

In order, the three test calls would return true, false, and true.

Change the regex haRegex to match the word Hazzah only when it has four or more letter z's.

let haStr = "Hazzzzah";

let haRegex = /Haz{3,}zah/; // Change this line

let result = haRegex.test(haStr);

\*\*\*\*\*\*\*

**Specify Exact Number of Matches**

You can specify the lower and upper number of patterns with quantity specifiers using curly brackets. Sometimes you only want a specific number of matches.

To specify a certain number of patterns, just have that one number between the curly brackets.

For example, to match only the word hah with the letter a 3 times, your regex would be /ha{3}h/.

let A4 = "haaaah";

let A3 = "haaah";

let A100 = "h" + "a".repeat(100) + "h";

let multipleHA = /ha{3}h/;

multipleHA.test(A4);

multipleHA.test(A3);

multipleHA.test(A100);

In order, the three test calls would return false, true, and false.

let timStr = "Timmmmber";

let timRegex = /Tim{4}ber/; // Change this line

let result = timRegex.test(timStr);

\*\*\*\*\*\*\*\*\*\*

**Check for All or None**

Sometimes the patterns you want to search for may have parts of it that may or may not exist. However, it may be important to check for them nonetheless.

You can specify the possible existence of an element with a question mark, ?. This checks for zero or one of the preceding element. You can think of this symbol as saying the previous element is optional.

For example, there are slight differences in American and British English and you can use the question mark to match both spellings.

let american = "color";

let british = "colour";

let rainbowRegex= /colou?r/;

rainbowRegex.test(american);

rainbowRegex.test(british);

Both uses of the test method would return true.

Change the regex favRegex to match both the American English (favorite) and the British English (favourite) version of the word.

let favWord = "favorite";

let favRegex = /favou?rite/; // Change this line

let result = favRegex.test(favWord);

\*\*\*\*\*\*

**Positive and Negative Lookahead**

Lookaheads are 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.

Lookaheads are a bit confusing but some examples will help.

let quit = "qu";

let noquit = "qt";

let quRegex= /q(?=u)/;

let qRegex = /q(?!u)/;

quit.match(quRegex);

noquit.match(qRegex);

Both of these match calls would return ["q"].

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);

Use lookaheads in the pwRegex to match passwords that are greater than 5 characters long, and have two consecutive digits.

let sampleWord = "astronaut";

let pwRegex = /^(?=\w{6})(?=\w+\d{2})/; // Change this line

let result = pwRegex.test(sampleWord);

\*\*\*\*\*\*\*

**Check For Mixed Grouping of Characters**

Sometimes we want to check for groups of characters using a Regular Expression and to achieve that we use parentheses ().

If you want to find either Penguin or Pumpkin in a string, you can use the following Regular Expression: /P(engu|umpk)in/g

Then check whether the desired string groups are in the test string by using the test() method.

let testStr = "Pumpkin";

let testRegex = /P(engu|umpk)in/;

testRegex.test(testStr);

The test method here would return true.

Fix the regex so that it checks for the names of Franklin Roosevelt or Eleanor Roosevelt in a case sensitive manner and it should make concessions for middle names.

Then fix the code so that the regex that you have created is checked against myString and either true or false is returned depending on whether the regex matches.

let myString = "Eleanor Roosevelt";

let myRegex = /(Franklin|Eleanor).\*Roosevelt/; // Change this line

let result = myRegex.test(myString); // Change this line

// After passing the challenge experiment with myString and see how the grouping works

\*\*\*\*\*\*

**Reuse Patterns Using 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 \1 to 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);

repeatStr.match(repeatRegex);

The test call would return true, and the match call would return ["regex regex", "regex"].

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

Use capture groups in reRegex to match a string that consists of only the same number repeated exactly three times separated by single spaces.

let repeatNum = "42 42 42";

let reRegex = /^(\d+)\s\1\s\1$/; // Change this line

let result = reRegex.test(repeatNum);

console.log(result);

\*\*\*\*\*\*\*

**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");

The replace call would return the string 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');

The replace call would return the string Camp Code.

Write a regex fixRegex using three capture groups that will search for each word in the string one two three. Then update the replaceText variable to replace one two three with the string three two one and assign the result to the result variable. Make sure you are utilizing capture groups in the replacement string using the dollar sign ($) syntax.

let str = "one two three";

let fixRegex = /(\w+)\s(\w+)\s(\w+)/; // Change this line

let replaceText = "$3 $2 $1";

let result = str.replace(fixRegex, replaceText);

console.log(result);

\*\*\*\*

**Remove Whitespace from Start and End**

Sometimes whitespace characters around strings are not wanted but are there. Typical processing of strings is to remove the whitespace at the start and end of it.

Write a regex and use the appropriate string methods to remove whitespace at the beginning and end of strings.

**Note:** The String.prototype.trim() method would work here, but you'll need to complete this challenge using regular expressions.

let hello = " Hello, World! ";

let wsRegex = /^\s\*(.\*\S)\s\*$/; // Change this line

let result = hello.replace(wsRegex, "$1"); // Change this line

**DEBUGGING**

**Use the JavaScript Console to Check the Value of a Variable**

Both Chrome and Firefox have excellent JavaScript consoles, also known as DevTools, for debugging your JavaScript.

You can find Developer tools in your Chrome's menu or Web Console in Firefox's menu. If you're using a different browser, or a mobile phone, we strongly recommend switching to desktop Firefox or Chrome.

The console.log() method, which "prints" the output of what's within its parentheses to the console, will likely be the most helpful debugging tool. Placing it at strategic points in your code can show you the intermediate values of variables. It's good practice to have an idea of what the output should be before looking at what it is. Having check points to see the status of your calculations throughout your code will help narrow down where the problem is.

Here's an example to print the string Hello world! to the console:

console.log('Hello world!');

Use the console.log() method to print the value of the variable a where noted in the code.

let a = 5;

let b = 1;

a++;

// Only change code below this line

let sumAB = a + b;

console.log(a);

\*\*\*\*\*\*\*\*\*\*\*

**Understanding the Differences between the freeCodeCamp and Browser Console**

You may have noticed that some freeCodeCamp JavaScript challenges include their own console. This console behaves a little differently than the browser console you used in the last challenge.

The following challenge is meant to highlight the main difference between the freeCodeCamp console and your browser console.

When you run ordinary JavaScript, the browser's console will display your console.log() statements the exact number of times it is called.

The freeCodeCamp console will print your console.log() statements a short time after the editor detects a change in the script, as well as during testing.

The freeCodeCamp console is cleared before the tests are run and, to avoid spam, only prints the logs during the first test (see the note below for exceptions).

If you would like to see every log for every test, run the tests, and open the browser console. If you prefer to use the browser console, and want it to mimic the freeCodeCamp console, place console.clear() before any other console calls, to clear the browser console.

**Note:** console.logs inside functions are printed to the freeCodeCamp console whenever those functions are called. This can help debugging functions that are called during testing.

First, use console.log to log the output variable. Then, use console.clear to clear the browser console.

// Open your browser console.

let output = "Get this to log once in the freeCodeCamp console and twice in the browser console";

// Use console.log() to print the output variable.

console.log(output);

// Run the tests to see the difference between the two consoles.

console.log(output);

// Now, add console.clear() before your console.log() to clear the browser console, and pass the tests.

console.clear();

console.log(output);

\*\*\*\*\*\*\*

**Use typeof to Check the Type of a Variable**

You can use typeof to check the data structure, or type, of a variable. This is useful in debugging when working with multiple data types. If you think you're adding two numbers, but one is actually a string, the results can be unexpected. Type errors can lurk in calculations or function calls. Be careful especially when you're accessing and working with external data in the form of a JavaScript Object Notation (JSON) object.

Here are some examples using typeof:

console.log(typeof "");

console.log(typeof 0);

console.log(typeof []);

console.log(typeof {});

In order, the console will display the strings string, number, object, and object.

JavaScript recognizes six primitive (immutable) data types: Boolean, Null, Undefined, Number, String, and Symbol (new with ES6) and one type for mutable items: Object. Note that in JavaScript, arrays are technically a type of object.

Add two console.log() statements to check the typeof each of the two variables seven and three in the code.

let seven = 7;

let three = "3";

console.log(seven + three);

// Only change code below this line

console.log(typeof seven);

console.log(typeof three);

Console output:

// running tests

// tests completed

// console output

73

number

string

73

number

string

\*\*\*\*\*\*\*\*

**Catch Misspelled Variable and Function Names**

The console.log() and typeof methods are the two primary ways to check intermediate values and types of program output. Now it's time to get into the common forms that bugs take. One syntax-level issue that fast typers can commiserate with is the humble spelling error.

Transposed, missing, or mis-capitalized characters in a variable or function name will have the browser looking for an object that doesn't exist - and complain in the form of a reference error. JavaScript variable and function names are case-sensitive.

Fix the two spelling errors in the code so the netWorkingCapital calculation works.

let receivables = 10;

let payables = 8;

let netWorkingCapital = receivables - payables;

console.log(`Net working capital is: ${netWorkingCapital}`);

\*\*\*

**Catch Unclosed Parentheses, Brackets, Braces and Quotes**

Another syntax error to be aware of is that all opening parentheses, brackets, curly braces, and quotes have a closing pair. Forgetting a piece tends to happen when you're editing existing code and inserting items with one of the pair types. Also, take care when nesting code blocks into others, such as adding a callback function as an argument to a method.

One way to avoid this mistake is as soon as the opening character is typed, immediately include the closing match, then move the cursor back between them and continue coding. Fortunately, most modern code editors generate the second half of the pair automatically.

Fix the two pair errors in the code.

let myArray = [1, 2, 3];

let arraySum = myArray.reduce((previous, current) => previous + current);

console.log(`Sum of array values is: ${arraySum}`);

\*\*\*

**Catch Mixed Usage of Single and Double Quotes**

JavaScript allows the use of both single (') and double (") quotes to declare a string. Deciding which one to use generally comes down to personal preference, with some exceptions.

Having two choices is great when a string has contractions or another piece of text that's in quotes. Just be careful that you don't close the string too early, which causes a syntax error.

Here are some examples of mixing quotes:

const grouchoContraction = "I've had a perfectly wonderful evening, but this wasn't it.";

const quoteInString = "Groucho Marx once said 'Quote me as saying I was mis-quoted.'";

const uhOhGroucho = 'I've had a perfectly wonderful evening, but this wasn't it.';

The first two are correct, but the third is incorrect.

Of course, it is okay to use only one style of quotes. You can escape the quotes inside the string by using the backslash (\) escape character:

const allSameQuotes = 'I\'ve had a perfectly wonderful evening, but this wasn\'t it.';

Fix the string so it either uses different quotes for the href value, or escape the existing ones. Keep the double quote marks around the entire string.

Run the Tests

Reset All Code

Get Help

let innerHtml = "<p>Click here to <a href='#Home'>return home</a></p>";

console.log(innerHtml);

\*\*\*\*

**Catch Use of Assignment Operator Instead of Equality Operator**

Branching programs, i.e. ones that do different things if certain conditions are met, rely on if, else if, and else statements in JavaScript. The condition sometimes takes the form of testing whether a result is equal to a value.

This logic is spoken (in English, at least) as "if x equals y, then ..." which can literally translate into code using the =, or assignment operator. This leads to unexpected control flow in your program.

As covered in previous challenges, the assignment operator (=) in JavaScript assigns a value to a variable name. And the == and === operators check for equality (the triple === tests for strict equality, meaning both value and type are the same).

The code below assigns x to be 2, which evaluates as true. Almost every value on its own in JavaScript evaluates to true, except what are known as the "falsy" values: false, 0, "" (an empty string), NaN, undefined, and null.

let x = 1;

let y = 2;

if (x = y) {

} else {

}

In this example, the code block within the if statement will run for any value of y, unless y is falsy. The else block, which we expect to run here, will not actually run.

Fix the condition so the program runs the right branch, and the appropriate value is assigned to result.

let x = 7;

let y = 9;

let result = "to come";

if(x == y) {

result = "Equal!";

} else {

result = "Not equal!";

}

console.log(result);

Console output:

Not equal!

\*\*\*\*\*\*\*\*\*\*

**Catch Missing Open and Closing Parenthesis After a Function Call**

When a function or method doesn't take any arguments, you may forget to include the (empty) opening and closing parentheses when calling it. Often times the result of a function call is saved in a variable for other use in your code. This error can be detected by logging variable values (or their types) to the console and seeing that one is set to a function reference, instead of the expected value the function returns.

The variables in the following example are different:

function myFunction() {

return "You rock!";

}

let varOne = myFunction;

let varTwo = myFunction();

Here varOne is the function myFunction, and varTwo is the string You rock!.

Fix the code so the variable result is set to the value returned from calling the function getNine.

function getNine() {

let x = 6;

let y = 3;

return x + y;

}

let result = getNine();

console.log(result);

Console output:

9

\*\*\*\*\*\*\*\*\*\*

**Catch Arguments Passed in the Wrong Order When Calling a Function**

Continuing the discussion on calling functions, the next bug to watch out for is when a function's arguments are supplied in the incorrect order. If the arguments are different types, such as a function expecting an array and an integer, this will likely throw a runtime error. If the arguments are the same type (all integers, for example), then the logic of the code won't make sense. Make sure to supply all required arguments, in the proper order to avoid these issues.

The function raiseToPower raises a base to an exponent. Unfortunately, it's not called properly - fix the code so the value of power is the expected 8.

function raiseToPower(b, e) {

return Math.pow(b, e);

}

let base = 2;

let exp = 3;

let power = raiseToPower(base, exp);

console.log(power);

\*\*\*\*\*

**Catch Off By One Errors When Using Indexing**

Off by one errors (sometimes called OBOE) crop up when you're trying to target a specific index of a string or array (to slice or access a segment), or when looping over the indices of them. JavaScript indexing starts at zero, not one, which means the last index is always one less than the length of the item. If you try to access an index equal to the length, the program may throw an "index out of range" reference error or print undefined.

When you use string or array methods that take index ranges as arguments, it helps to read the documentation and understand if they are inclusive (the item at the given index is part of what's returned) or not. Here are some examples of off by one errors:

let alphabet = "abcdefghijklmnopqrstuvwxyz";

let len = alphabet.length;

for (let i = 0; i <= len; i++) {

console.log(alphabet[i]);

}

for (let j = 1; j < len; j++) {

console.log(alphabet[j]);

}

for (let k = 0; k < len; k++) {

console.log(alphabet[k]);

}

The first example here loops one too many times, and the second loops one too few times (missing the first index, 0). The third example is correct.

Fix the two indexing errors in the following function so all the numbers 1 through 5 are printed to the console.

function countToFive() {

let firstFive = "12345";

let len = firstFive.length;

// Only change code below this line

for (let i = 0; i < len; i++) {

// Only change code above this line

console.log(firstFive[i]);

}

}

countToFive();

Console output:

// running tests

// tests completed

// console output

1

2

3

4

5

\*\*\*\*\*\*\*\*

**Use Caution When Reinitializing Variables Inside a Loop**

Sometimes it's necessary to save information, increment counters, or re-set variables within a loop. A potential issue is when variables either should be reinitialized, and aren't, or vice versa. This is particularly dangerous if you accidentally reset the variable being used for the terminal condition, causing an infinite loop.

Printing variable values with each cycle of your loop by using console.log() can uncover buggy behavior related to resetting, or failing to reset a variable.

The following function is supposed to create a two-dimensional array with m rows and n columns of zeroes. Unfortunately, it's not producing the expected output because the row variable isn't being reinitialized (set back to an empty array) in the outer loop. Fix the code so it returns a correct 3x2 array of zeroes, which looks like [[0, 0], [0, 0], [0, 0]].

function zeroArray(m, n) {

// Creates a 2-D array with m rows and n columns of zeroes

let newArray = [];

let row = [];

for (let i = 0; i < m; i++) {

// Adds the m-th row into newArray

row=[];

for (let j = 0; j < n; j++) {

// Pushes n zeroes into the current row to create the columns

row.push(0);

}

// Pushes the current row, which now has n zeroes in it, to the array

newArray.push(row);

}

return newArray;

}

let matrix = zeroArray(3, 2);

console.log(matrix);

\*\*\*\*\*\*\*\*\*\*\*\*

**Prevent Infinite Loops with a Valid Terminal Condition**

The final topic is the dreaded infinite loop. Loops are great tools when you need your program to run a code block a certain number of times or until a condition is met, but they need a terminal condition that ends the looping. Infinite loops are likely to freeze or crash the browser, and cause general program execution mayhem, which no one wants.

There was an example of an infinite loop in the introduction to this section - it had no terminal condition to break out of the while loop inside loopy(). Do NOT call this function!

function loopy() {

while(true) {

console.log("Hello, world!");

}

}

It's the programmer's job to ensure that the terminal condition, which tells the program when to break out of the loop code, is eventually reached. One error is incrementing or decrementing a counter variable in the wrong direction from the terminal condition. Another one is accidentally resetting a counter or index variable within the loop code, instead of incrementing or decrementing it.

The myFunc() function contains an infinite loop because the terminal condition i != 4 will never evaluate to false (and break the looping) - i will increment by 2 each pass, and jump right over 4 since i is odd to start. Fix the comparison operator in the terminal condition so the loop only runs for i less than or equal to 4.

function myFunc() {

for (let i = 1; i <= 4; i += 2) {

console.log("Still going!");

}

}

\*\*\*\*\*\*\*\*

**Use an Array to Store a Collection of Data**

The below is an example of the simplest implementation of an array data structure. This is known as a one-dimensional array, meaning it only has one level, or that it does not have any other arrays nested within it. Notice it contains booleans, strings, and numbers, among other valid JavaScript data types:

let simpleArray = ['one', 2, 'three', true, false, undefined, null];

console.log(simpleArray.length);

The console.log call displays 7.

All arrays have a length property, which as shown above, can be very easily accessed with the syntax Array.length. A more complex implementation of an array can be seen below. This is known as a multi-dimensional array, or an array that contains other arrays. Notice that this array also contains JavaScript objects, which we will examine very closely in our next section, but for now, all you need to know is that arrays are also capable of storing complex objects.

let complexArray = [

[

{

one: 1,

two: 2

},

{

three: 3,

four: 4

}

],

[

{

a: "a",

b: "b"

},

{

c: "c",

d: "d"

}

]

];

We have defined a variable called yourArray. Complete the statement by assigning an array of at least 5 elements in length to the yourArray variable. Your array should contain at least one string, one number, and one boolean.

let yourArray=[1,'HELLO',true,false,"e"]; // Change this line

\*\*\*\*\*\*\*\*

**Access an Array's Contents Using Bracket Notation**

The fundamental feature of any data structure is, of course, the ability to not only store data, but to be able to retrieve that data on command. So, now that we've learned how to create an array, let's begin to think about how we can access that array's information.

When we define a simple array as seen below, there are 3 items in it:

let ourArray = ["a", "b", "c"];

In an array, each array item has an index. This index doubles as the position of that item in the array, and how you reference it. However, it is important to note, that JavaScript arrays are zero-indexed, meaning that the first element of an array is actually at the ***zeroth*** position, not the first. In order to retrieve an element from an array we can enclose an index in brackets and append it to the end of an array, or more commonly, to a variable which references an array object. This is known as bracket notation. For example, if we want to retrieve the a from ourArray and assign it to a variable, we can do so with the following code:

let ourVariable = ourArray[0];

Now ourVariable has the value of a.

In addition to accessing the value associated with an index, you can also *set* an index to a value using the same notation:

ourArray[1] = "not b anymore";

Using bracket notation, we have now reset the item at index 1 from the string b, to not b anymore. Now ourArray is ["a", "not b anymore", "c"].

In order to complete this challenge, set the 2nd position (index 1) of myArray to anything you want, besides the letter b.

let myArray = ["a", "b", "c", "d"];

// Only change code below this line

myArray[1]="e"

// Only change code above this line

console.log(myArray);

Console output:

// running tests

// tests completed

// console output

[ 'a', 'e', 'c', 'd' ]

[ 'a', 'e', 'c', 'd' ]

\*\*\*\*\*\*\*\*

**Add Items to an Array with push() and unshift()**

An array's length, like the data types it can contain, is not fixed. Arrays can be defined with a length of any number of elements, and elements can be added or removed over time; in other words, arrays are mutable. In this challenge, we will look at two methods with which we can programmatically modify an array: Array.push() and Array.unshift().

Both methods take one or more elements as parameters and add those elements to the array the method is being called on; the push() method adds elements to the end of an array, and unshift() adds elements to the beginning. Consider the following:

let twentyThree = 'XXIII';

let romanNumerals = ['XXI', 'XXII'];

romanNumerals.unshift('XIX', 'XX');

romanNumerals would have the value ['XIX', 'XX', 'XXI', 'XXII'].

romanNumerals.push(twentyThree);

romanNumerals would have the value ['XIX', 'XX', 'XXI', 'XXII', 'XXIII']. Notice that we can also pass variables, which allows us even greater flexibility in dynamically modifying our array's data.

We have defined a function, mixedNumbers, which we are passing an array as an argument. Modify the function by using push() and unshift() to add 'I', 2, 'three' to the beginning of the array and 7, 'VIII', 9 to the end so that the returned array contains representations of the numbers 1-9 in order.

function mixedNumbers(arr) {

// Only change code below this line

arr.push(7,'VIII', 9);

arr.unshift('I', 2, 'three');

// Only change code above this line

return arr;

}

console.log(mixedNumbers(['IV', 5, 'six']));

Console output:

// running tests

// tests completed

// console output

[ 'I', 2, 'three', 'IV', 5, 'six', 7, 'VIII', 9 ]

[ 'I', 2, 'three', 'IV', 5, 'six', 7, 'VIII', 9 ]

\*\*\*

**Remove Items from an Array with pop() and shift()**

Both push() and unshift() have corresponding methods that are nearly functional opposites: pop() and shift(). As you may have guessed by now, instead of adding, pop() *removes* an element from the end of an array, while shift() removes an element from the beginning. The key difference between pop() and shift() and their cousins push() and unshift(), is that neither method takes parameters, and each only allows an array to be modified by a single element at a time.

Let's take a look:

let greetings = ['whats up?', 'hello', 'see ya!'];

greetings.pop();

greetings would have the value ['whats up?', 'hello'].

greetings.shift();

greetings would have the value ['hello'].

We can also return the value of the removed element with either method like this:

let popped = greetings.pop();

greetings would have the value [], and popped would have the value hello.

We have defined a function, popShift, which takes an array as an argument and returns a new array. Modify the function, using pop() and shift(), to remove the first and last elements of the argument array, and assign the removed elements to their corresponding variables, so that the returned array contains their values.

function popShift(arr) {

let popped=arr.pop(); // Change this line

let shifted=arr.shift(); // Change this line

return [shifted, popped];

}

console.log(popShift(['challenge', 'is', 'not', 'complete']));

\*\*\*\*\*\*\*\*\*

**Remove Items Using splice()**

Ok, so we've learned how to remove elements from the beginning and end of arrays using shift() and pop(), but what if we want to remove an element from somewhere in the middle? Or remove more than one element at once? Well, that's where splice() comes in. splice() allows us to do just that: **remove any number of consecutive elements** from anywhere in an array.

splice() can take up to 3 parameters, but for now, we'll focus on just the first 2. The first two parameters of splice() are integers which represent indexes, or positions, of the array that splice() is being called upon. And remember, arrays are *zero-indexed*, so to indicate the first element of an array, we would use 0. splice()'s first parameter represents the index on the array from which to begin removing elements, while the second parameter indicates the number of elements to delete. For example:

let array = ['today', 'was', 'not', 'so', 'great'];

array.splice(2, 2);

Here we remove 2 elements, beginning with the third element (at index 2). array would have the value ['today', 'was', 'great'].

splice() not only modifies the array it's being called on, but it also returns a new array containing the value of the removed elements:

let array = ['I', 'am', 'feeling', 'really', 'happy'];

let newArray = array.splice(3, 2);

newArray has the value ['really', 'happy'].

We've initialized an array arr. Use splice() to remove elements from arr, so that it only contains elements that sum to the value of 10.

const arr = [2, 4, 5, 1, 7, 5, 2, 1];

// Only change code below this line

arr.splice(1,4);

// Only change code above this line

console.log(arr);

Console output:

// running tests

// tests completed

// console output

[ 2, 5, 2, 1 ]

[ 2, 5, 2, 1 ]

\*\*\*\*\*\*\*\*\*\*

**Add Items Using splice()**

Remember in the last challenge we mentioned that splice() can take up to three parameters? Well, you can use the third parameter, comprised of one or more element(s), to add to the array. This can be incredibly useful for quickly switching out an element, or a set of elements, for another.

const numbers = [10, 11, 12, 12, 15];

const startIndex = 3;

const amountToDelete = 1;

numbers.splice(startIndex, amountToDelete, 13, 14);

console.log(numbers);

The second occurrence of 12 is removed, and we add 13 and 14 at the same index. The numbers array would now be [ 10, 11, 12, 13, 14, 15 ].

Here, we begin with an array of numbers. Then, we pass the following to splice(): The index at which to begin deleting elements (3), the number of elements to be deleted (1), and the remaining arguments (13, 14) will be inserted starting at that same index. Note that there can be any number of elements (separated by commas) following amountToDelete, each of which gets inserted.

We have defined a function, htmlColorNames, which takes an array of HTML colors as an argument. Modify the function using splice() to remove the first two elements of the array and add 'DarkSalmon' and 'BlanchedAlmond' in their respective places.

function htmlColorNames(arr) {

// Only change code below this line

arr.splice(0,2,'DarkSalmon','BlanchedAlmond');

// Only change code above this line

return arr;z

}

console.log(htmlColorNames(['DarkGoldenRod', 'WhiteSmoke', 'LavenderBlush', 'PaleTurquoise', 'FireBrick']));

// running tests

// tests completed

// console output

[ 'DarkSalmon',

'BlanchedAlmond',

'LavenderBlush',

'PaleTurquoise',

'FireBrick' ]

[ 'DarkSalmon',

'BlanchedAlmond',

'LavenderBlush',

'PaleTurquoise',

'FireBrick' ]

**Copy Array Items Using slice()**

The next method we will cover is slice(). Rather than modifying an array, slice() copies or *extracts* a given number of elements to a new array, leaving the array it is called upon untouched. slice() takes only 2 parameters — the first is the index at which to begin extraction, and the second is the index at which to stop extraction (extraction will occur up to, but not including the element at this index). Consider this:

let weatherConditions = ['rain', 'snow', 'sleet', 'hail', 'clear'];

let todaysWeather = weatherConditions.slice(1, 3);

todaysWeather would have the value ['snow', 'sleet'], while weatherConditions would still have ['rain', 'snow', 'sleet', 'hail', 'clear'].

In effect, we have created a new array by extracting elements from an existing array.

We have defined a function, forecast, that takes an array as an argument. Modify the function using slice() to extract information from the argument array and return a new array that contains the string elements warm and sunny.

function forecast(arr) {

// Only change code below this line

arr=arr.slice(2,4);

return arr;

}

// Only change code above this line

console.log(forecast(['cold', 'rainy', 'warm', 'sunny', 'cool', 'thunderstorms']));

// running tests

// tests completed

// console output

[ 'warm', 'sunny' ]

[ 'warm', 'sunny' ]

\*\*\*\*\*\*

**Copy an Array with the Spread Operator**

While slice() allows us to be selective about what elements of an array to copy, among several other useful tasks, ES6's new spread operator allows us to easily copy *all* of an array's elements, in order, with a simple and highly readable syntax. The spread syntax simply looks like this: ...

In practice, we can use the spread operator to copy an array like so:

let thisArray = [true, true, undefined, false, null];

let thatArray = [...thisArray];

thatArray equals [true, true, undefined, false, null]. thisArray remains unchanged and thatArray contains the same elements as thisArray.

We have defined a function, copyMachine which takes arr (an array) and num (a number) as arguments. The function is supposed to return a new array made up of num copies of arr. We have done most of the work for you, but it doesn't work quite right yet. Modify the function using spread syntax so that it works correctly (hint: another method we have already covered might come in handy here!).

function copyMachine(arr, num) {

let newArr = [];

while (num >= 1) {

// Only change code below this line

newArr.push([...arr]);

// Only change code above this line

num--;

}

return newArr;

}

console.log(copyMachine([true, false, true], 2));

// running tests

// tests completed

// console output

[ [ true, false, true ], [ true, false, true ] ]

[ [ true, false, true ], [ true, false, true ] ]

\*\*\*\*\*

**Combine Arrays with the Spread Operator**

Another huge advantage of the spread operator is the ability to combine arrays, or to insert all the elements of one array into another, at any index. With more traditional syntaxes, we can concatenate arrays, but this only allows us to combine arrays at the end of one, and at the start of another. Spread syntax makes the following operation extremely simple:

let thisArray = ['sage', 'rosemary', 'parsley', 'thyme'];

let thatArray = ['basil', 'cilantro', ...thisArray, 'coriander'];

thatArray would have the value ['basil', 'cilantro', 'sage', 'rosemary', 'parsley', 'thyme', 'coriander'].

Using spread syntax, we have just achieved an operation that would have been more complex and more verbose had we used traditional methods.

We have defined a function spreadOut that returns the variable sentence. Modify the function using the spread operator so that it returns the array ['learning', 'to', 'code', 'is', 'fun'].

function spreadOut() {

let fragment = ['to', 'code'];

let sentence=['learning',...fragment,'is','fun']; // Change this line

return sentence;

}

console.log(spreadOut());

// running tests

// tests completed

// console output

[ 'learning', 'to', 'code', 'is', 'fun' ]

[ 'learning', 'to', 'code', 'is', 'fun' ]

**Check For The Presence of an Element With indexOf()**

Since arrays can be changed, or *mutated*, at any time, there's no guarantee about where a particular piece of data will be on a given array, or if that element even still exists. Luckily, JavaScript provides us with another built-in method, indexOf(), that allows us to quickly and easily check for the presence of an element on an array. indexOf() takes an element as a parameter, and when called, it returns the position, or index, of that element, or -1 if the element does not exist on the array.

For example:

let fruits = ['apples', 'pears', 'oranges', 'peaches', 'pears'];

fruits.indexOf('dates');

fruits.indexOf('oranges');

fruits.indexOf('pears');

indexOf('dates') returns -1, indexOf('oranges') returns 2, and indexOf('pears') returns 1 (the first index at which each element exists).

indexOf() can be incredibly useful for quickly checking for the presence of an element on an array. We have defined a function, quickCheck, that takes an array and an element as arguments. Modify the function using indexOf() so that it returns true if the passed element exists on the array, and false if it does not.

function quickCheck(arr, elem) {

// Only change code below this line

if(arr.indexOf(elem)>=0){

return true;

}

else{

return false;

}

// Only change code above this line

}

console.log(quickCheck(['squash', 'onions', 'shallots'], 'mushrooms'));

// running tests

false

// tests completed

// console output

false

\*\*\*\*\*\*\*\*

**Iterate Through All an Array's Items Using For Loops**

Sometimes when working with arrays, it is very handy to be able to iterate through each item to find one or more elements that we might need, or to manipulate an array based on which data items meet a certain set of criteria. JavaScript offers several built in methods that each iterate over arrays in slightly different ways to achieve different results (such as every(), forEach(), map(), etc.), however the technique which is most flexible and offers us the greatest amount of control is a simple for loop.

Consider the following:

function greaterThanTen(arr) {

let newArr = [];

for (let i = 0; i < arr.length; i++) {

if (arr[i] > 10) {

newArr.push(arr[i]);

}

}

return newArr;

}

greaterThanTen([2, 12, 8, 14, 80, 0, 1]);

Using a for loop, this function iterates through and accesses each element of the array, and subjects it to a simple test that we have created. In this way, we have easily and programmatically determined which data items are greater than 10, and returned a new array, [12, 14, 80], containing those items.

We have defined a function, filteredArray, which takes arr, a nested array, and elem as arguments, and returns a new array. elem represents an element that may or may not be present on one or more of the arrays nested within arr. Modify the function, using a for loop, to return a filtered version of the passed array such that any array nested within arr containing elem has been removed.

function filteredArray(arr, elem) {

let newArr = [];

// Only change code below this line

for (let i = 0; i < arr.length; i++) {

if (arr[i].indexOf(elem) === -1) {

newArr.push(arr[i]);

}

}

// Only change code above this line

return newArr;

}

console.log(filteredArray([[3, 2, 3], [1, 6, 3], [3, 13, 26], [19, 3, 9]], 3));

// running tests

// tests completed

// console output

[]

[]

\*\*\*\*\*\*\*\*\*\*\*\*

**Create complex multi-dimensional arrays**

Awesome! You have just learned a ton about arrays! This has been a fairly high level overview, and there is plenty more to learn about working with arrays, much of which you will see in later sections. But before moving on to looking at Objects, lets take one more look, and see how arrays can become a bit more complex than what we have seen in previous challenges.

One of the most powerful features when thinking of arrays as data structures, is that arrays can contain, or even be completely made up of other arrays. We have seen arrays that contain arrays in previous challenges, but fairly simple ones. However, arrays can contain an infinite depth of arrays that can contain other arrays, each with their own arbitrary levels of depth, and so on. In this way, an array can very quickly become very complex data structure, known as a multi-dimensional, or nested array. Consider the following example:

let nestedArray = [

['deep'],

[

['deeper'], ['deeper']

],

[

[

['deepest'], ['deepest']

],

[

[

['deepest-est?']

]

]

]

];

The deep array is nested 2 levels deep. The deeper arrays are 3 levels deep. The deepest arrays are 4 levels, and the deepest-est? is 5.

While this example may seem convoluted, this level of complexity is not unheard of, or even unusual, when dealing with large amounts of data. However, we can still very easily access the deepest levels of an array this complex with bracket notation:

console.log(nestedArray[2][1][0][0][0]);

This logs the string deepest-est?. And now that we know where that piece of data is, we can reset it if we need to:

nestedArray[2][1][0][0][0] = 'deeper still';

console.log(nestedArray[2][1][0][0][0]);

Now it logs deeper still.

We have defined a variable, myNestedArray, set equal to an array. Modify myNestedArray, using any combination of strings, numbers, and booleans for data elements, so that it has exactly five levels of depth (remember, the outer-most array is level 1). Somewhere on the third level, include the string deep, on the fourth level, include the string deeper, and on the fifth level, include the string deepest.

let myNestedArray = [ //level 1

// change code below this line

['unshift', false, 1, 2, 3, 'complex', 'nested'], //level 2

[

['loop', 'shift', 6, 7, 1000, 'method', 'deep'], //level 3

[

['concat', false, true, 'deeper', 'spread', 'array'], //level 4

[

['deepest', 'mutate', 1327.98, 'splice', 'slice', 'push'] //level 5

]

]

],

['iterate', 1.3849, 7, '8.4876', 'arbitrary', 'depth']

// change code above this line

];

\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Add Key-Value Pairs to JavaScript Objects**

At their most basic, objects are just collections of key-value pairs. In other words, they are pieces of data (values) mapped to unique identifiers called properties (keys). Take a look at an example:

const tekkenCharacter = {

player: 'Hwoarang',

fightingStyle: 'Tae Kwon Doe',

human: true

};

The above code defines a Tekken video game character object called tekkenCharacter. It has three properties, each of which map to a specific value. If you want to add an additional property, such as "origin", it can be done by assigning origin to the object:

tekkenCharacter.origin = 'South Korea';

This uses dot notation. If you were to observe the tekkenCharacter object, it will now include the origin property. Hwoarang also had distinct orange hair. You can add this property with bracket notation by doing:

tekkenCharacter['hair color'] = 'dyed orange';

Bracket notation is required if your property has a space in it or if you want to use a variable to name the property. In the above case, the property is enclosed in quotes to denote it as a string and will be added exactly as shown. Without quotes, it will be evaluated as a variable and the name of the property will be whatever value the variable is. Here's an example with a variable:

const eyes = 'eye color';

tekkenCharacter[eyes] = 'brown';

After adding all the examples, the object will look like this:

{

player: 'Hwoarang',

fightingStyle: 'Tae Kwon Doe',

human: true,

origin: 'South Korea',

'hair color': 'dyed orange',

'eye color': 'brown'

};

A foods object has been created with three entries. Using the syntax of your choice, add three more entries to it: bananas with a value of 13, grapes with a value of 35, and strawberries with a value of 27.

let foods = {

apples: 25,

oranges: 32,

plums: 28

};

// Only change code below this line

foods['bananas']=13;

foods['grapes']=35;

foods.strawberries=27;

// Only change code above this line

console.log(foods);

// running tests

{ apples: 25,

oranges: 32,

plums: 28,

bananas: 13,

grapes: 35,

strawberries: 27 }

// tests completed

// console output

{ apples: 25,

oranges: 32,

plums: 28,

bananas: 13,

grapes: 35,

strawberries: 27 }

\*\*\*\*\*\*\*\*\*

**Modify an Object Nested Within an Object**

Now let's take a look at a slightly more complex object. Object properties can be nested to an arbitrary depth, and their values can be any type of data supported by JavaScript, including arrays and even other objects. Consider the following:

let nestedObject = {

id: 28802695164,

date: 'December 31, 2016',

data: {

totalUsers: 99,

online: 80,

onlineStatus: {

active: 67,

away: 13,

busy: 8

}

}

};

nestedObject has three properties: id (value is a number), date (value is a string), and data (value is an object with its nested structure). While structures can quickly become complex, we can still use the same notations to access the information we need. To assign the value 10 to the busy property of the nested onlineStatus object, we use dot notation to reference the property:

nestedObject.data.onlineStatus.busy = 10;

Here we've defined an object userActivity, which includes another object nested within it. Set the value of the online key to 45.

let userActivity = {

id: 23894201352,

date: 'January 1, 2017',

data: {

totalUsers: 51,

online: 42

}

};

// Only change code below this line

userActivity.data['online']=45;

// Only change code above this line

console.log(userActivity);

\*\*\*\*\*\*\*\*\*\*

**Access Property Names with Bracket Notation**

In the first object challenge we mentioned the use of bracket notation as a way to access property values using the evaluation of a variable. For instance, imagine that our foods object is being used in a program for a supermarket cash register. We have some function that sets the selectedFood and we want to check our foods object for the presence of that food. This might look like:

let selectedFood = getCurrentFood(scannedItem);

let inventory = foods[selectedFood];

This code will evaluate the value stored in the selectedFood variable and return the value of that key in the foods object, or undefined if it is not present. Bracket notation is very useful because sometimes object properties are not known before runtime or we need to access them in a more dynamic way.

We've defined a function, checkInventory, which receives a scanned item as an argument. Return the current value of the scannedItem key in the foods object. You can assume that only valid keys will be provided as an argument to checkInventory.

let foods = {

apples: 25,

oranges: 32,

plums: 28,

bananas: 13,

grapes: 35,

strawberries: 27

};

function checkInventory(scannedItem) {

// Only change code below this line

return foods.[scannedItem];

// Only change code above this line

}

console.log(checkInventory("apples"));

25

\*\*\*\*

**Use the delete Keyword to Remove Object Properties**

Now you know what objects are and their basic features and advantages. In short, they are key-value stores which provide a flexible, intuitive way to structure data, ***and***, they provide very fast lookup time. Throughout the rest of these challenges, we will describe several common operations you can perform on objects so you can become comfortable applying these useful data structures in your programs.

In earlier challenges, we have both added to and modified an object's key-value pairs. Here we will see how we can *remove* a key-value pair from an object.

Let's revisit our foods object example one last time. If we wanted to remove the apples key, we can remove it by using the delete keyword like this:

delete foods.apples;

Use the delete keyword to remove the oranges, plums, and strawberries keys from the foods object.

let foods = {

apples: 25,

oranges: 32,

plums: 28,

bananas: 13,

grapes: 35,

strawberries: 27

};

// Only change code below this line

delete foods['oranges'];

delete foods['plums'];

delete foods['strawberries'];

// Only change code above this line

console.log(foods);

\*\*\*\*\*\*\*\*\*\*

**Check if an Object has a Property**

Now we can add, modify, and remove keys from objects. But what if we just wanted to know if an object has a specific property? JavaScript provides us with two different ways to do this. One uses the hasOwnProperty() method and the other uses the in keyword. If we have an object users with a property of Alan, we could check for its presence in either of the following ways:

users.hasOwnProperty('Alan');

'Alan' in users;

Both of these would return true.

Finish writing the function so that it returns true if the object passed to it contains all four names, Alan, Jeff, Sarah and Ryan and returns false otherwise.

let users = {

Alan: {

age: 27,

online: true

},

Jeff: {

age: 32,

online: true

},

Sarah: {

age: 48,

online: true

},

Ryan: {

age: 19,

online: true

}

};

function isEveryoneHere(userObj) {

// Only change code below this line

if (userObj.hasOwnProperty('Alan')&&userObj.hasOwnProperty('Jeff')&&userObj.hasOwnProperty('Sarah')&&userObj.hasOwnProperty('Ryan')) {

return true;

} else {

return false;

}

// Only change code above this line

}

console.log(isEveryoneHere(users));

\*\*\*\*\*\*\*\*\*\*

**Iterate Through the Keys of an Object with a for...in Statement**

Sometimes you may need to iterate through all the keys within an object. This requires a specific syntax in JavaScript called a for...in statement. For our users object, this could look like:

for (let user in users) {

console.log(user);

}

This would log Alan, Jeff, Sarah, and Ryan - each value on its own line.

In this statement, we defined a variable user, and as you can see, this variable was reset during each iteration to each of the object's keys as the statement looped through the object, resulting in each user's name being printed to the console.

**NOTE:** Objects do not maintain an ordering to stored keys like arrays do; thus a key's position on an object, or the relative order in which it appears, is irrelevant when referencing or accessing that key.

We've defined a function countOnline which accepts one argument (a users object). Use a for...in statement within this function to loop through the users object passed into the function and return the number of users whose online property is set to true. An example of a users object which could be passed to countOnline is shown below. Each user will have an online property with either a true or false value.

{

Alan: {

online: false

},

Jeff: {

online: true

},

Sarah: {

online: false

}

}

let users = {

Alan: {

age: 27,

online: false

},

Jeff: {

age: 32,

online: true

},

Sarah: {

age: 48,

online: false

},

Ryan: {

age: 19,

online: true

}

};

function countOnline(obj) {

// change code below this line

let result = 0;

for (let user in obj) {

if (obj[user].online === true) {

result++;

}

}

return result;

// change code above this line

}

console.log(countOnline(users));

\*\*\*\*\*\*\*\*\*\*

**Generate an Array of All Object Keys with Object.keys()**

We can also generate an array which contains all the keys stored in an object using the Object.keys() method and passing in an object as the argument. This will return an array with strings representing each property in the object. Again, there will be no specific order to the entries in the array.

Finish writing the getArrayOfUsers function so that it returns an array containing all the properties in the object it receives as an argument.

let users = {

Alan: {

age: 27,

online: false

},

Jeff: {

age: 32,

online: true

},

Sarah: {

age: 48,

online: false

},

Ryan: {

age: 19,

online: true

}

};

function getArrayOfUsers(obj) {

// Only change code below this line

return Object.keys(obj);

;

// Only change code above this line

}

console.log(getArrayOfUsers(users));

\*\*\*\*\*\*

**Modify an Array Stored in an Object**

Now you've seen all the basic operations for JavaScript objects. You can add, modify, and remove key-value pairs, check if keys exist, and iterate over all the keys in an object. As you continue learning JavaScript you will see even more versatile applications of objects. Additionally, the Data Structures lessons located in the Coding Interview Prep section of the curriculum also cover the ES6 Map and Set objects, both of which are similar to ordinary objects but provide some additional features. Now that you've learned the basics of arrays and objects, you're fully prepared to begin tackling more complex problems using JavaScript!

Take a look at the object we've provided in the code editor. The user object contains three keys. The data key contains five keys, one of which contains an array of friends. From this, you can see how flexible objects are as data structures. We've started writing a function addFriend. Finish writing it so that it takes a user object and adds the name of the friend argument to the array stored in user.data.friends and returns that array.

let user = {

name: 'Kenneth',

age: 28,

data: {

username: 'kennethCodesAllDay',

joinDate: 'March 26, 2016',

organization: 'freeCodeCamp',

friends: [

'Sam',

'Kira',

'Tomo'

],

location: {

city: 'San Francisco',

state: 'CA',

country: 'USA'

}

}

};

function addFriend(userObj, friend) {

// Only change code below this line

userObj.data.friends.push(friend);

return user.data.friends;

// Only change code above this line

}

console.log(addFriend(user, 'Pete'));

\*\*\*\*\*

**BASIC ALGORITHM SCRIPTING**

**Convert Celsius to Fahrenheit**

The algorithm to convert from Celsius to Fahrenheit is the temperature in Celsius times 9/5, plus 32.

You are given a variable celsius representing a temperature in Celsius. Use the variable fahrenheit already defined and assign it the Fahrenheit temperature equivalent to the given Celsius temperature. Use the algorithm mentioned above to help convert the Celsius temperature to Fahrenheit.

function convertToF(celsius) {

let fahrenheit;

fahrenheit= (celsius\*9/5)+32;

return fahrenheit;

}

convertToF(30);

\*\*\*\*\*

**Reverse a String**

Reverse the provided string.

You may need to turn the string into an array before you can reverse it.

Your result must be a string.

function reverseString(str) {

return str.split('').reverse().join('');

return str;

}

reverseString("hello");

\*\*\*\*\*\*

**Factorialize a Number**

Return the factorial of the provided integer.

If the integer is represented with the letter n, a factorial is the product of all positive integers less than or equal to n.

Factorials are often represented with the shorthand notation n!

For example: 5! = 1 \* 2 \* 3 \* 4 \* 5 = 120

Only integers greater than or equal to zero will be supplied to the function.

Run the Tests

Reset All Code

Get Help

function factorialize(num) {

let total=1;

for(let i=1;i<=num;i++){

total \*=i;

}

return total;

}

factorialize(5);

|  |  |
| --- | --- |
| function factorialize(num) { |  |
|  | // Create a variable named total and set it equal to 1. |
|  | // This will account for the 0 edge case, because when |
|  | // num = 0, the loop will not run at all, and total will |
|  | // be returned as 1 at the end of the function. |
|  | let total = 1 |
|  | // Start the for loop at 1. |
|  | // Run the loop as long as i is less than or equal to num. |
|  | // (So when num = 0, i is never less or equal to num, and the |
|  | // loop never runs). |
|  | // Increment i by one each time the loop runs. |
|  | for (let i = 1; i <= num; i++) { |
|  | // Multiply the total by i until we reach num. |
|  | // Save the product as the variable total each time. |
|  | // This can also be written: total = total \* i. |
|  | total \*= i |
|  | // total i |
|  | // 1st run: total = 1 \* 1 total = 1 |
|  | // 2nd run: total = 1 \* 2 total = 2 |
|  | // 3rd run: total = 2 \* 3 total = 6 |
|  | // 4th run: total = 6 \* 4 total = 24 |
|  | // 5th run: total = 24 \* 5 total = 120 |
|  | } |
|  | // return total as our answer |
|  | return total; |
|  | } |
|  |  |
|  | factorialize(5); |

\*\*\*\*\*

**Find the Longest Word in a String**

Return the length of the longest word in the provided sentence.

Your response should be a number.

function findLongestWordLength(str) {

let sortedWords = str.split(' ').sort((a, b) => b.length - a.length)

return sortedWords[0].length

}

findLongestWordLength("The quick brown fox jumped over the lazy dog");

\*\*\*\*

**Return Largest Numbers in Arrays**

Return an array consisting of the largest number from each provided sub-array. For simplicity, the provided array will contain exactly 4 sub-arrays.

Remember, you can iterate through an array with a simple for loop, and access each member with array syntax arr[i].

function largestOfFour(arr) {

let result=[];

for(let i=0;i<arr.length;i++){

let largestnumber=arr[i][0];

for(let j=0;j<arr[i].length;j++){

if(arr[i][j]>largestnumber){

largestnumber=arr[i][j];

}

}

result[i]=largestnumber;

}

return result;

}

largestOfFour([[4, 5, 1, 3], [13, 27, 18, 26], [32, 35, 37, 39], [1000, 1001, 857, 1]]);

\*\*\*\*\*\*\*

**Confirm the Ending**

Check if a string (first argument, str) ends with the given target string (second argument, target).

This challenge *can* be solved with the .endsWith() method, which was introduced in ES2015. But for the purpose of this challenge, we would like you to use one of the JavaScript substring methods instead.

function confirmEnding(str, target) {

var newStr = '';

newStr = str.substring(str.length - target.length);

return newStr === target;

}

confirmEnding("Bastian", "n");

\*\*\*\*\*\*\*\*

**Repeat a String Repeat a String**

Repeat a given string str (first argument) for num times (second argument). Return an empty string if num is not a positive number. For the purpose of this challenge, do *not* use the built-in .repeat() method.

function repeatStringNumTimes(str, num) {

var repeatedstring='';

while(num>0){

repeatedstring +=str;

num--;

}

return repeatedstring;

}

repeatStringNumTimes("abc", 3);

\*\*\*\*\*\*

**Truncate a String**

Truncate a string (first argument) if it is longer than the given maximum string length (second argument). Return the truncated string with a ... ending.

function truncateString(str, num) {

if (str.length > num) {

return str.slice(0, num) + "...";

} else {

return str;

}

}

\*\*\*\*\*\*\*

**Finders Keepers**

Create a function that looks through an array arr and returns the first element in it that passes a 'truth test'. This means that given an element x, the 'truth test' is passed if func(x) is true. If no element passes the test, return undefined.

function findElement(arr, func) {

let num = 0;

for (let i = 0; i < arr.length; i++) {

num = arr[i];

if (func(num)) {

return num;

}

}

return undefined;

}

\*\*\*\*\*\*

**Boo who**

Check if a value is classified as a boolean primitive. Return true or false.

Boolean primitives are true and false.

function booWho(bool) {

if(bool===true ||bool===false){

return true;

}

else{

return false;

}

}

console.log(booWho(null));

\*\*\*\*\*

**Title Case a Sentence**

Return the provided string with the first letter of each word capitalized. Make sure the rest of the word is in lower case.

For the purpose of this exercise, you should also capitalize connecting words like the and of.

function titleCase(str) {

var arr = str.split(' ');

var newStr = '';

for (var i = 0; i < arr.length; i++) {

var lower = arr[i].toLowerCase();

newStr += lower.charAt(0).toUpperCase() + lower.slice(1) + ' ';

}

return newStr.trim();

}

\*\*\*\*\*\*\*

**Slice and Splice**

You are given two arrays and an index.

Copy each element of the first array into the second array, in order.

Begin inserting elements at index n of the second array.

Return the resulting array. The input arrays should remain the same after the function runs.

function frankenSplice(arr1, arr2, n) {

let combinedArrays=arr2.slice();

combinedArrays.splice(n,0,...arr1);

return combinedArrays;

}

frankenSplice([1, 2, 3], [4, 5, 6], 1);

\*\*\*\*\*\*

**Falsy Bouncer**

Remove all falsy values from an array.

Falsy values in JavaScript are false, null, 0, "", undefined, and NaN.

Hint: Try converting each value to a Boolean.

function bouncer(arr) {

// Don't show a false ID to this bouncer.

return arr.filter(Boolean);

}

bouncer([7, "ate", "", false, 9]);

\*\*\*\*\*\*\*\*

**Where do I Belong**

Return the lowest index at which a value (second argument) should be inserted into an array (first argument) once it has been sorted. The returned value should be a number.

For example, getIndexToIns([1,2,3,4], 1.5) should return 1 because it is greater than 1 (index 0), but less than 2 (index 1).

Likewise, getIndexToIns([20,3,5], 19) should return 2 because once the array has been sorted it will look like [3,5,20] and 19 is less than 20 (index 2) and greater than 5 (index 1).

function getIndexToIns(arr, num) {

if(arr.indexOf(num) === -1) {

arr.push(num);

}

var newArr = arr;

function sortNumber(a,b) {

return a - b;

}

newArr.sort(sortNumber);

return newArr.indexOf(num);

}

getIndexToIns([3, 10, 5], 3);

\*\*\*\*\*\*\*\*

**Mutations**

Return true if the string in the first element of the array contains all of the letters of the string in the second element of the array.

For example, ["hello", "Hello"], should return true because all of the letters in the second string are present in the first, ignoring case.

The arguments ["hello", "hey"] should return false because the string hello does not contain a y.

Lastly, ["Alien", "line"], should return true because all of the letters in line are present in Alien.

function mutation(arr) {

var item1 = arr[0].toLowerCase();

var item2 = arr[1].toLowerCase();

for (var i = 0; i < item2.length; i++) {

var match = item1.indexOf(item2[i]);

if (match === -1) {

return false;

}

}

return true;

}

mutation(["hello", "hey"]);

\*\*\*\*\*\*\*\*

**Chunky Monkey**

Write a function that splits an array (first argument) into groups the length of size (second argument) and returns them as a two-dimensional array.

function chunkArrayInGroups(arr, size) {

var masterArr = [];

var item = 0;

while(item < arr.length) {

masterArr.push(arr.slice(item, item += size));

}

return masterArr;

}

chunkArrayInGroups(["a", "b", "c", "d"], 2);

\*\*\*\*\*\*\*\*

OBJECT ORIENTED PROGRAMMING

**Create a Basic JavaScript Object**

Think about things people see every day, like cars, shops, and birds. These are all objects: tangible things people can observe and interact with.

What are some qualities of these objects? A car has wheels. Shops sell items. Birds have wings.

These qualities, or properties, define what makes up an object. Note that similar objects share the same properties, but may have different values for those properties. For example, all cars have wheels, but not all cars have the same number of wheels.

Objects in JavaScript are used to model real-world objects, giving them properties and behavior just like their real-world counterparts. Here's an example using these concepts to create a duck object:

let duck = {

name: "Aflac",

numLegs: 2

};

This duck object has two property/value pairs: a name of Aflac and a numLegs of 2.

Create a dog object with name and numLegs properties, and set them to a string and a number, respectively.

let dog = {

name:"JACk",

numLegs:4

};

\*\*\*\*\*

**Use Dot Notation to Access the Properties of an Object**

The last challenge created an object with various properties. Now you'll see how to access the values of those properties. Here's an example:

let duck = {

name: "Aflac",

numLegs: 2

};

console.log(duck.name);

Dot notation is used on the object name, duck, followed by the name of the property, name, to access the value of Aflac.

Print both properties of the dog object to your console.

let dog = {

name: "Spot",

numLegs: 4

};

console.log(dog.name)

console.log(dog.numLegs)

// Only change code below this line

\*\*\*\*\*

**Create a Method on an Object**

Objects can have a special type of property, called a method.

Methods are properties that are functions. This adds different behavior to an object. Here is the duck example with a method:

let duck = {

name: "Aflac",

numLegs: 2,

sayName: function() {return "The name of this duck is " + duck.name + ".";}

};

duck.sayName();

The example adds the sayName method, which is a function that returns a sentence giving the name of the duck. Notice that the method accessed the name property in the return statement using duck.name. The next challenge will cover another way to do this.

Using the dog object, give it a method called sayLegs. The method should return the sentence This dog has 4 legs.

let dog = {

name: "Spot",

numLegs: 4,

sayLegs:function(){

return "This dog has "+ dog.numLegs +" legs.";

}

};

dog.sayLegs();

\*\*\*\*\*

**Make Code More Reusable with the this Keyword**

The last challenge introduced a method to the duck object. It used duck.name dot notation to access the value for the name property within the return statement:

sayName: function() {return "The name of this duck is " + duck.name + ".";}

While this is a valid way to access the object's property, there is a pitfall here. If the variable name changes, any code referencing the original name would need to be updated as well. In a short object definition, it isn't a problem, but if an object has many references to its properties there is a greater chance for error.

A way to avoid these issues is with the this keyword:

let duck = {

name: "Aflac",

numLegs: 2,

sayName: function() {return "The name of this duck is " + this.name + ".";}

};

this is a deep topic, and the above example is only one way to use it. In the current context, this refers to the object that the method is associated with: duck. If the object's name is changed to mallard, it is not necessary to find all the references to duck in the code. It makes the code reusable and easier to read.

Modify the dog.sayLegs method to remove any references to dog. Use the duck example for guidance.

let dog = {

name: "Spot",

numLegs: 4,

sayLegs: function() {return "This dog has " + this.numLegs + " legs.";}

};

dog.sayLegs();

\*\*\*\*\*

**Define a Constructor Function**

Constructors are functions that create new objects. They define properties and behaviors that will belong to the new object. Think of them as a blueprint for the creation of new objects.

Here is an example of a constructor:

function Bird() {

this.name = "Albert";

this.color = "blue";

this.numLegs = 2;

}

This constructor defines a Bird object with properties name, color, and numLegs set to Albert, blue, and 2, respectively. Constructors follow a few conventions:

* Constructors are defined with a capitalized name to distinguish them from other functions that are not constructors.
* Constructors use the keyword this to set properties of the object they will create. Inside the constructor, this refers to the new object it will create.
* Constructors define properties and behaviors instead of returning a value as other functions might.

Create a constructor, Dog, with properties name, color, and numLegs that are set to a string, a string, and a number, respectively.

Run the Tests

Reset All Code

Get Help

function Dog(){

this.name="jack";

this.color="black";

this.numLegs=4;

}

\*\*\*\*\*

**Use a Constructor to Create Objects**

Here's the Bird constructor from the previous challenge:

function Bird() {

this.name = "Albert";

this.color = "blue";

this.numLegs = 2;

}

let blueBird = new Bird();

**NOTE:** this inside the constructor always refers to the object being created.

Notice that the new operator is used when calling a constructor. This tells JavaScript to create a new instance of Bird called blueBird. Without the new operator, this inside the constructor would not point to the newly created object, giving unexpected results. Now blueBird has all the properties defined inside the Bird constructor:

blueBird.name;

blueBird.color;

blueBird.numLegs;

Just like any other object, its properties can be accessed and modified:

blueBird.name = 'Elvira';

blueBird.name;

Use the Dog constructor from the last lesson to create a new instance of Dog, assigning it to a variable hound.

function Dog() {

this.name = "Rupert";

this.color = "brown";

this.numLegs = 4;

}

// Only change code below this line

let hound=new Dog();

//for my testing purpose

console.log(hound.name);

console.log(hound.color);

console.log(hound.numLegs);

Rupert

brown

4

Now it is clear that new operator is used to call the constructor new variable which is assigned to it will have same property that it is defined to it.let hound=new Dog();

So hound will have all objects property which is defined in the Dog constructor.

\*\*\*\*\*\*\*

**Extend Constructors to Receive Arguments**

The Bird and Dog constructors from the last challenge worked well. However, notice that all Birds that are created with the Bird constructor are automatically named Albert, are blue in color, and have two legs. What if you want birds with different values for name and color? It's possible to change the properties of each bird manually but that would be a lot of work:

let swan = new Bird();

swan.name = "Carlos";

swan.color = "white";

Suppose you were writing a program to keep track of hundreds or even thousands of different birds in an aviary. It would take a lot of time to create all the birds, then change the properties to different values for every one. To more easily create different Bird objects, you can design your Bird constructor to accept parameters:

function Bird(name, color) {

this.name = name;

this.color = color;

this.numLegs = 2;

}

Then pass in the values as arguments to define each unique bird into the Bird constructor: let cardinal = new Bird("Bruce", "red"); This gives a new instance of Bird with name and color properties set to Bruce and red, respectively. The numLegs property is still set to 2. The cardinal has these properties:

cardinal.name

cardinal.color

cardinal.numLegs

The constructor is more flexible. It's now possible to define the properties for each Bird at the time it is created, which is one way that JavaScript constructors are so useful. They group objects together based on shared characteristics and behavior and define a blueprint that automates their creation.

Create another Dog constructor. This time, set it up to take the parameters name and color, and have the property numLegs fixed at 4. Then create a new Dog saved in a variable terrier. Pass it two strings as arguments for the name and color properties.

function Dog(name,color) {

this.name=name;

this.color=color;

this.numLegs=4;

}

let terrier=new Dog('JACK','BLACK');

//for testing purpose

console.log(terrier.name);

console.log(terrier.color);

console.log(terrier.numLegs);

Console output:

JACK

BLACK

4

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*

**Verify an Object's Constructor with instanceof**

Anytime a constructor function creates a new object, that object is said to be an instance of its constructor. JavaScript gives a convenient way to verify this with the instanceof operator. instanceof allows you to compare an object to a constructor, returning true or false based on whether or not that object was created with the constructor. Here's an example:

let Bird = function(name, color) {

this.name = name;

this.color = color;

this.numLegs = 2;

}

let crow = new Bird("Alexis", "black");

crow instanceof Bird;

This instanceof method would return true.

If an object is created without using a constructor, instanceof will verify that it is not an instance of that constructor:

let canary = {

name: "Mildred",

color: "Yellow",

numLegs: 2

};

canary instanceof Bird;

This instanceof method would return false.

Create a new instance of the House constructor, calling it myHouse and passing a number of bedrooms. Then, use instanceof to verify that it is an instance of House.

function House(numBedrooms) {

this.numBedrooms = 4;

}

// Only change code below this line

let myHouse=new House();

myHouse instanceof House;

\*\*\*\*\*

**Understand Own Properties**

In the following example, the Bird constructor defines two properties: name and numLegs:

function Bird(name) {

this.name = name;

this.numLegs = 2;

}

let duck = new Bird("Donald");

let canary = new Bird("Tweety");

name and numLegs are called own properties, because they are defined directly on the instance object. That means that duck and canary each has its own separate copy of these properties. In fact every instance of Bird will have its own copy of these properties. The following code adds all of the own properties of duck to the array ownProps:

let ownProps = [];

for (let property in duck) {

if(duck.hasOwnProperty(property)) {

ownProps.push(property);

}

}

console.log(ownProps);

The console would display the value ["name", "numLegs"].

Add the own properties of canary to the array ownProps.

function Bird(name) {

this.name = name;

this.numLegs = 2;

}

let canary = new Bird("Tweety");

let ownProps = [];

// Only change code below this line

for(let property in canary){

if(canary.hasOwnProperty(property)){

ownProps.push(property);

}

}

\*\*\*\*\*\*\*

**Use Prototype Properties to Reduce Duplicate Code**

Since numLegs will probably have the same value for all instances of Bird, you essentially have a duplicated variable numLegs inside each Bird instance.

This may not be an issue when there are only two instances, but imagine if there are millions of instances. That would be a lot of duplicated variables.

A better way is to use Bird’s prototype. Properties in the prototype are shared among ALL instances of Bird. Here's how to add numLegs to the Bird prototype:

Bird.prototype.numLegs = 2;

Now all instances of Bird have the numLegs property.

console.log(duck.numLegs);

console.log(canary.numLegs);

Since all instances automatically have the properties on the prototype, think of a prototype as a "recipe" for creating objects. Note that the prototype for duck and canary is part of the Bird constructor as Bird.prototype. Nearly every object in JavaScript has a prototype property which is part of the constructor function that created it.

Add a numLegs property to the prototype of Dog

function Dog(name) {

this.name = name;

}

Dog.prototype.numLegs=4;

// Only change code above this line

let beagle = new Dog("Snoopy");

\*\*\*\*\*\*\*

**Iterate Over All Properties**

You have now seen two kinds of properties: own properties and prototype properties. Own properties are defined directly on the object instance itself. And prototype properties are defined on the prototype.

function Bird(name) {

this.name = name; //own property

}

Bird.prototype.numLegs = 2; // prototype property

let duck = new Bird("Donald");

Here is how you add duck's own properties to the array ownProps and prototype properties to the array prototypeProps:

let ownProps = [];

let prototypeProps = [];

for (let property in duck) {

if(duck.hasOwnProperty(property)) {

ownProps.push(property);

} else {

prototypeProps.push(property);

}

}

console.log(ownProps);

console.log(prototypeProps);

console.log(ownProps) would display ["name"] in the console, and console.log(prototypeProps) would display ["numLegs"].

Add all of the own properties of beagle to the array ownProps. Add all of the prototype properties of Dog to the array prototypeProps.

function Dog(name) {

this.name = name;

}

Dog.prototype.numLegs = 4;

let beagle = new Dog("Snoopy");

let ownProps = [];

let prototypeProps = [];

// Only change code below this line

for(let property in beagle){

if(beagle.hasOwnProperty(property)){

ownProps.push(property);

}else{

prototypeProps.push(property);

}

}

\*\*\*\*\*\*\*

**Understand the Constructor Property**

There is a special constructor property located on the object instances duck and beagle that were created in the previous challenges:

let duck = new Bird();

let beagle = new Dog();

console.log(duck.constructor === Bird);

console.log(beagle.constructor === Dog);

Both of these console.log calls would display true in the console.

Note that the constructor property is a reference to the constructor function that created the instance. The advantage of the constructor property is that it's possible to check for this property to find out what kind of object it is. Here's an example of how this could be used:

function joinBirdFraternity(candidate) {

if (candidate.constructor === Bird) {

return true;

} else {

return false;

}

}

**Note:** Since the constructor property can be overwritten (which will be covered in the next two challenges) it’s generally better to use the instanceof method to check the type of an object.

Write a joinDogFraternity function that takes a candidate parameter and, using the constructor property, return true if the candidate is a Dog, otherwise return false.

function Dog(name) {

this.name = name;

}

// Only change code below this line

function joinDogFraternity(candidate) {

if(candidate.constructor===Dog){

return true;

}else{

return false;

}

}

\*\*\*\*\*\*

**Change the Prototype to a New Object**

Up until now you have been adding properties to the prototype individually:

Bird.prototype.numLegs = 2;

This becomes tedious after more than a few properties.

Bird.prototype.eat = function() {

console.log("nom nom nom");

}

Bird.prototype.describe = function() {

console.log("My name is " + this.name);

}

A more efficient way is to set the prototype to a new object that already contains the properties. This way, the properties are added all at once:

Bird.prototype = {

numLegs: 2,

eat: function() {

console.log("nom nom nom");

},

describe: function() {

console.log("My name is " + this.name);

}

};

Add the property numLegs and the two methods eat() and describe() to the prototype of Dog by setting the prototype to a new object.

function Dog(name) {

this.name = name;

}

Dog.prototype = {

// Only change code below this line

numLegs:4,

eat:function(){

console.log("bone");

},

describe:function(){

console.log("my name is "+Dog.name+".");

}

};

//testing purpose

console.log(Dog.prototype.eat());

console.log(Dog.prototype.describe());

// running tests

// tests completed

// console output

bone

undefined

my name is Dog.

undefined

\*\*\*\*\*\*\*

**Remember to Set the Constructor Property when Changing the Prototype**

There is one crucial side effect of manually setting the prototype to a new object. It erases the constructor property! This property can be used to check which constructor function created the instance, but since the property has been overwritten, it now gives false results:

duck.constructor === Bird;

duck.constructor === Object;

duck instanceof Bird;

In order, these expressions would evaluate to false, true, and true.

To fix this, whenever a prototype is manually set to a new object, remember to define the constructor property:

Bird.prototype = {

constructor: Bird,

numLegs: 2,

eat: function() {

console.log("nom nom nom");

},

describe: function() {

console.log("My name is " + this.name);

}

};

Define the constructor property on the Dog prototype.

function Dog(name) {

this.name = name;

}

// Only change code below this line

Dog.prototype = {

constructor: Dog,

numLegs: 4,

eat: function() {

console.log("nom nom nom");

},

describe: function() {

console.log("My name is " + this.name);

}

};

\*\*\*\*\*\*\*

**Understand Where an Object’s Prototype Comes From**

Just like people inherit genes from their parents, an object inherits its prototype directly from the constructor function that created it. For example, here the Bird constructor creates the duck object:

function Bird(name) {

this.name = name;

}

let duck = new Bird("Donald");

duck inherits its prototype from the Bird constructor function. You can show this relationship with the isPrototypeOf method:

Bird.prototype.isPrototypeOf(duck);

This would return true.

Use isPrototypeOf to check the prototype of beagle.

function Dog(name) {

this.name = name;

}

let beagle = new Dog("Snoopy");

// Only change code below this line

Dog.prototype.isPrototypeOf(beagle);

\*\*\*\*\*

**Understand the Prototype Chain**

All objects in JavaScript (with a few exceptions) have a prototype. Also, an object’s prototype itself is an object.

function Bird(name) {

this.name = name;

}

typeof Bird.prototype;

Because a prototype is an object, a prototype can have its own prototype! In this case, the prototype of Bird.prototype is Object.prototype:

Object.prototype.isPrototypeOf(Bird.prototype);

How is this useful? You may recall the hasOwnProperty method from a previous challenge:

let duck = new Bird("Donald");

duck.hasOwnProperty("name");

The hasOwnProperty method is defined in Object.prototype, which can be accessed by Bird.prototype, which can then be accessed by duck. This is an example of the prototype chain. In this prototype chain, Bird is the supertype for duck, while duck is the subtype. Object is a supertype for both Bird and duck. Object is a supertype for all objects in JavaScript. Therefore, any object can use the hasOwnProperty method.

Modify the code to show the correct prototype chain.

function Dog(name) {

this.name = name;

}

let beagle = new Dog("Snoopy");

Dog.prototype.isPrototypeOf(beagle); // yields true

// Fix the code below so that it evaluates to true

Object.prototype.isPrototypeOf(Dog.prototype);

\*\*\*\*\*\*

**Use Inheritance So You Don't Repeat Yourself**

There's a principle in programming called Don't Repeat Yourself (DRY). The reason repeated code is a problem is because any change requires fixing code in multiple places. This usually means more work for programmers and more room for errors.

Notice in the example below that the describe method is shared by Bird and Dog:

Bird.prototype = {

constructor: Bird,

describe: function() {

console.log("My name is " + this.name);

}

};

Dog.prototype = {

constructor: Dog,

describe: function() {

console.log("My name is " + this.name);

}

};

The describe method is repeated in two places. The code can be edited to follow the DRY principle by creating a supertype (or parent) called Animal:

function Animal() { };

Animal.prototype = {

constructor: Animal,

describe: function() {

console.log("My name is " + this.name);

}

};

Since Animal includes the describe method, you can remove it from Bird and Dog:

Bird.prototype = {

constructor: Bird

};

Dog.prototype = {

constructor: Dog

};

The eat method is repeated in both Cat and Bear. Edit the code in the spirit of DRY by moving the eat method to the Animal supertype.

function Cat(name) {

this.name = name;

}

Cat.prototype = {

constructor: Cat,

};

function Bear(name) {

this.name = name;

}

Bear.prototype = {

constructor: Bear,

};

function Animal() { }

Animal.prototype = {

constructor: Animal,

eat: function() {

console.log("nom nom nom");

}

};

\*\*\*\*\*\*

**Inherit Behaviors from a Supertype**

In the previous challenge, you created a supertype called Animal that defined behaviors shared by all animals:

function Animal() { }

Animal.prototype.eat = function() {

console.log("nom nom nom");

};

This and the next challenge will cover how to reuse Animal's methods inside Bird and Dog without defining them again. It uses a technique called inheritance. This challenge covers the first step: make an instance of the supertype (or parent). You already know one way to create an instance of Animal using the new operator:

let animal = new Animal();

There are some disadvantages when using this syntax for inheritance, which are too complex for the scope of this challenge. Instead, here's an alternative approach without those disadvantages:

let animal = Object.create(Animal.prototype);

Object.create(obj) creates a new object, and sets obj as the new object's prototype. Recall that the prototype is like the "recipe" for creating an object. By setting the prototype of animal to be Animal's prototype, you are effectively giving the animal instance the same "recipe" as any other instance of Animal.

animal.eat();

animal instanceof Animal;

The instanceof method here would return true.

Use Object.create to make two instances of Animal named duck and beagle.

function Animal() { }

Animal.prototype = {

constructor: Animal,

eat: function() {

console.log("nom nom nom");

}

};

// Only change code below this line

let duck=Object.create(Animal.prototype); // Change this line

let beagle=Object.create(Animal.prototype); // Change this line

//testing purpose

// use can craete like this also "let animal = new Animal();" but it has many disadvantages so create it like it

//let duck=Object.create(Animal.prototype); // Change this line

//let beagle=Object.create(Animal.prototype); // Change this line

console.log(duck.eat());

nom nom nom

undefined

\*\*\*\*\*\*

**Set the Child's Prototype to an Instance of the Parent**

In the previous challenge you saw the first step for inheriting behavior from the supertype (or parent) Animal: making a new instance of Animal.

This challenge covers the next step: set the prototype of the subtype (or child)—in this case, Bird—to be an instance of Animal.

Bird.prototype = Object.create(Animal.prototype);

Remember that the prototype is like the "recipe" for creating an object. In a way, the recipe for Bird now includes all the key "ingredients" from Animal.

let duck = new Bird("Donald");

duck.eat();

duck inherits all of Animal's properties, including the eat method.

Modify the code so that instances of Dog inherit from Animal.

Run the Tests

Reset All Code

Get Help

function Animal() { }

Animal.prototype = {

constructor: Animal,

eat: function() {

console.log("nom nom nom");

}

};

function Dog() { }

// Only change code below this line

Dog.prototype=Object.create(Animal.prototype);

let beagle = new Dog();

\*\*\*\*\*\*

**Reset an Inherited Constructor Property**

When an object inherits its prototype from another object, it also inherits the supertype's constructor property.

Here's an example:

function Bird() { }

Bird.prototype = Object.create(Animal.prototype);

let duck = new Bird();

duck.constructor

But duck and all instances of Bird should show that they were constructed by Bird and not Animal. To do so, you can manually set Bird's constructor property to the Bird object:

Bird.prototype.constructor = Bird;

duck.constructor

Fix the code so duck.constructor and beagle.constructor return their respective constructors.

function Animal() { }

function Bird() { }

function Dog() { }

Bird.prototype = Object.create(Animal.prototype);

Dog.prototype = Object.create(Animal.prototype);

// Only change code below this line

Bird.prototype.constructor=Bird;

Dog.prototype.constructor=Dog;

let duck = new Bird();

let beagle = new Dog();

///testing

console.log(duck.constructor);

console.log(Dog.constructor);

console.log(Bird.constructor);

console.log(beagle.constructor);

console.log(Animal.constructor);

[Function: Bird]

[Function: Function]

[Function: Function]

[Function: Dog]

[Function: Function]

\*\*\*\*\*

**Add Methods After Inheritance**

A constructor function that inherits its prototype object from a supertype constructor function can still have its own methods in addition to inherited methods.

For example, Bird is a constructor that inherits its prototype from Animal:

function Animal() { }

Animal.prototype.eat = function() {

console.log("nom nom nom");

};

function Bird() { }

Bird.prototype = Object.create(Animal.prototype);

Bird.prototype.constructor = Bird;

In addition to what is inherited from Animal, you want to add behavior that is unique to Bird objects. Here, Bird will get a fly() function. Functions are added to Bird's prototype the same way as any constructor function:

Bird.prototype.fly = function() {

console.log("I'm flying!");

};

Now instances of Bird will have both eat() and fly() methods:

let duck = new Bird();

duck.eat();

duck.fly();

duck.eat() would display the string nom nom nom in the console, and duck.fly() would display the string I'm flying!.

Add all necessary code so the Dog object inherits from Animal and the Dog's prototype constructor is set to Dog. Then add a bark() method to the Dog object so that beagle can both eat() and bark(). The bark() method should print Woof! to the console.

function Animal() { }

Animal.prototype.eat = function() { console.log("nom nom nom"); };

function Dog() { }

// Only change code below this line

Dog.prototype=Object.create(Animal.prototype);

Dog.prototype.constructor=Dog;

Dog.prototype.bark=function(){

console.log("Woof!")

}

// Only change code above this line

let beagle = new Dog();

//testing

console.log(beagle.eat());

console.log(beagle.bark());

nom nom nom

undefined

Woof!

undefined

\*\*\*\*\*

**Override Inherited Methods**

In previous lessons, you learned that an object can inherit its behavior (methods) from another object by referencing its prototype object:

ChildObject.prototype = Object.create(ParentObject.prototype);

Then the ChildObject received its own methods by chaining them onto its prototype:

ChildObject.prototype.methodName = function() {...};

It's possible to override an inherited method. It's done the same way - by adding a method to ChildObject.prototype using the same method name as the one to override. Here's an example of Bird overriding the eat() method inherited from Animal:

function Animal() { }

Animal.prototype.eat = function() {

return "nom nom nom";

};

function Bird() { }

Bird.prototype = Object.create(Animal.prototype);

Bird.prototype.eat = function() {

return "peck peck peck";

};

If you have an instance let duck = new Bird(); and you call duck.eat(), this is how JavaScript looks for the method on duck’s prototype chain:

1. duck => Is eat() defined here? No.
2. Bird => Is eat() defined here? => Yes. Execute it and stop searching.
3. Animal => eat() is also defined, but JavaScript stopped searching before reaching this level.
4. Object => JavaScript stopped searching before reaching this level.

Override the fly() method for Penguin so that it returns the string Alas, this is a flightless bird.

function Bird() { }

Bird.prototype.fly = function() { return "I am flying!"; };

function Penguin() { }

Penguin.prototype = Object.create(Bird.prototype);

Penguin.prototype.constructor = Penguin;

// Only change code below this line

Penguin.prototype.fly=function(){

return "Alas, this is a flightless bird."

}

// Only change code above this line

let penguin = new Penguin();

console.log(penguin.fly());

// running tests

// tests completed

// console output

Alas, this is a flightless bird.

Alas, this is a flightless bird.

\*\*\*\*\*

**Use a Mixin to Add Common Behavior Between Unrelated Objects**

As you have seen, behavior is shared through inheritance. However, there are cases when inheritance is not the best solution. Inheritance does not work well for unrelated objects like Bird and Airplane. They can both fly, but a Bird is not a type of Airplane and vice versa.

For unrelated objects, it's better to use mixins. A mixin allows other objects to use a collection of functions.

let flyMixin = function(obj) {

obj.fly = function() {

console.log("Flying, wooosh!");

}

};

The flyMixin takes any object and gives it the fly method.

let bird = {

name: "Donald",

numLegs: 2

};

let plane = {

model: "777",

numPassengers: 524

};

flyMixin(bird);

flyMixin(plane);

Here bird and plane are passed into flyMixin, which then assigns the fly function to each object. Now bird and plane can both fly:

bird.fly();

plane.fly();

The console would display the string Flying, wooosh! twice, once for each .fly() call.

Note how the mixin allows for the same fly method to be reused by unrelated objects bird and plane.

Create a mixin named glideMixin that defines a method named glide. Then use the glideMixin to give both bird and boat the ability to glide.

let bird = {

name: "Donald",

numLegs: 2

};

let boat = {

name: "Warrior",

type: "race-boat"

};

// Only change code below this line

let glideMixin=function(obj){

obj.glide=function(){

console.log("lets glide");

}

}

glideMixin(bird);

glideMixin(boat);

bird.glide();

boat.glide();

// running tests

// tests completed

// console output

lets glide

lets glide

\*\*\*\*\*\*

**Use Closure to Protect Properties Within an Object from Being Modified Externally**

In the previous challenge, bird had a public property name. It is considered public because it can be accessed and changed outside of bird's definition.

bird.name = "Duffy";

Therefore, any part of your code can easily change the name of bird to any value. Think about things like passwords and bank accounts being easily changeable by any part of your codebase. That could cause a lot of issues.

The simplest way to make this public property private is by creating a variable within the constructor function. This changes the scope of that variable to be within the constructor function versus available globally. This way, the variable can only be accessed and changed by methods also within the constructor function.

function Bird() {

let hatchedEgg = 10;

this.getHatchedEggCount = function() {

return hatchedEgg;

};

}

let ducky = new Bird();

ducky.getHatchedEggCount();

Here getHatchedEggCount is a privileged method, because it has access to the private variable hatchedEgg. This is possible because hatchedEgg is declared in the same context as getHatchedEggCount. In JavaScript, a function always has access to the context in which it was created. This is called closure.

Change how weight is declared in the Bird function so it is a private variable. Then, create a method getWeight that returns the value of weight 15.

function Bird() {

let weight = 15;

this.getWeight = function() {

return weight;

};

}

\*\*\*\*

**Understand the Immediately Invoked Function Expression (IIFE)**

A common pattern in JavaScript is to execute a function as soon as it is declared:

(function () {

console.log("Chirp, chirp!");

})();

This is an anonymous function expression that executes right away, and outputs Chirp, chirp! immediately.

Note that the function has no name and is not stored in a variable. The two parentheses () at the end of the function expression cause it to be immediately executed or invoked. This pattern is known as an immediately invoked function expression or IIFE.

Rewrite the function makeNest and remove its call so instead it's an anonymous immediately invoked function expression (IIFE).

Question:

function makeNest() {

console.log("A cozy nest is ready");

}

makeNest();

Answer:

(function () {

console.log("A cozy nest is ready");

})

();

// running tests

// tests completed

// console output

A cozy nest is ready

A cozy nest is ready

\*\*\*\*\*

**Use an IIFE to Create a Module**

An immediately invoked function expression (IIFE) is often used to group related functionality into a single object or module. For example, an earlier challenge defined two mixins:

function glideMixin(obj) {

obj.glide = function() {

console.log("Gliding on the water");

};

}

function flyMixin(obj) {

obj.fly = function() {

console.log("Flying, wooosh!");

};

}

We can group these mixins into a module as follows:

let motionModule = (function () {

return {

glideMixin: function(obj) {

obj.glide = function() {

console.log("Gliding on the water");

};

},

flyMixin: function(obj) {

obj.fly = function() {

console.log("Flying, wooosh!");

};

}

}

})();

Note that you have an immediately invoked function expression (IIFE) that returns an object motionModule. This returned object contains all of the mixin behaviors as properties of the object. The advantage of the module pattern is that all of the motion behaviors can be packaged into a single object that can then be used by other parts of your code. Here is an example using it:

motionModule.glideMixin(duck);

duck.glide();

Create a module named funModule to wrap the two mixins isCuteMixin and singMixin. funModule should return an object.

Question:

let isCuteMixin = function(obj) {

obj.isCute = function() {

return true;

};

};

let singMixin = function(obj) {

obj.sing = function() {

console.log("Singing to an awesome tune");

};

};

Answer:

let funModule =( function(){

return{

isCuteMixin : function(obj) {

obj.isCute = function() {

return true;

};

},

singMixin:function(obj) {

obj.sing = function() {

console.log("Singing to an awesome tune");

};

}

}

})

();

\*\*\*\*\*

**Functional Programming**

**Learn About Functional Programming**

Functional programming is a style of programming where solutions are simple, isolated functions, without any side effects outside of the function scope: INPUT -> PROCESS -> OUTPUT

Functional programming is about:

1. Isolated functions - there is no dependence on the state of the program, which includes global variables that are subject to change
2. Pure functions - the same input always gives the same output
3. Functions with limited side effects - any changes, or mutations, to the state of the program outside the function are carefully controlled

The members of freeCodeCamp happen to love tea.

In the code editor, the prepareTea and getTea functions are already defined for you. Call the getTea function to get 40 cups of tea for the team, and store them in the tea4TeamFCC variable.

Run the Tests

Reset All Code

Get Help

Passed

The tea4TeamFCC variable should hold 40 cups of tea for the team.

Passed

The tea4TeamFCC variable should hold cups of green tea.

// Function that returns a string representing a cup of green tea

const prepareTea = () => 'greenTea';

/\*

Given a function (representing the tea type) and number of cups needed, the

following function returns an array of strings (each representing a cup of

a specific type of tea).

\*/

const getTea = (numOfCups) => {

const teaCups = [];

for(let cups = 1; cups <= numOfCups; cups += 1) {

const teaCup = prepareTea();

teaCups.push(teaCup);

}

return teaCups;

};

// Only change code below this line

const tea4TeamFCC = getTea(40);

// Only change code above this line

\*\*\*\*\*\*\*\*

**Understand Functional Programming Terminology**

The FCC Team had a mood swing and now wants two types of tea: green tea and black tea. General Fact: Client mood swings are pretty common.

With that information, we'll need to revisit the getTea function from last challenge to handle various tea requests. We can modify getTea to accept a function as a parameter to be able to change the type of tea it prepares. This makes getTea more flexible, and gives the programmer more control when client requests change.

But first, let's cover some functional terminology:

Callbacks are the functions that are slipped or passed into another function to decide the invocation of that function. You may have seen them passed to other methods, for example in filter, the callback function tells JavaScript the criteria for how to filter an array.

Functions that can be assigned to a variable, passed into another function, or returned from another function just like any other normal value, are called first class functions. In JavaScript, all functions are first class functions.

The functions that take a function as an argument, or return a function as a return value are called higher order functions.

When functions are passed in to or returned from another function, then those functions which were passed in or returned can be called a lambda.

Prepare 27 cups of green tea and 13 cups of black tea and store them in tea4GreenTeamFCC and tea4BlackTeamFCC variables, respectively. Note that the getTea function has been modified so it now takes a function as the first argument.

Note: The data (the number of cups of tea) is supplied as the last argument. We'll discuss this more in later lessons.

// Function that returns a string representing a cup of green tea

const prepareGreenTea = () => 'greenTea';

// Function that returns a string representing a cup of black tea

const prepareBlackTea = () => 'blackTea';

/\*

Given a function (representing the tea type) and number of cups needed, the

following function returns an array of strings (each representing a cup of

a specific type of tea).

\*/

const getTea = (prepareTea, numOfCups) => {

const teaCups = [];

for(let cups = 1; cups <= numOfCups; cups += 1) {

const teaCup = prepareTea();

teaCups.push(teaCup);

}

return teaCups;

};

// Only change code below this line

const tea4GreenTeamFCC = getTea(prepareGreenTea,27);

const tea4BlackTeamFCC = getTea(prepareBlackTea,13);;

// Only change code above this line

console.log(

tea4GreenTeamFCC,

tea4BlackTeamFCC

);

\*\*\*\*\*\*\*\*\*

**Understand the Hazards of Using Imperative Code**

Functional programming is a good habit. It keeps your code easy to manage, and saves you from sneaky bugs. But before we get there, let's look at an imperative approach to programming to highlight where you may have issues.

In English (and many other languages), the imperative tense is used to give commands. Similarly, an imperative style in programming is one that gives the computer a set of statements to perform a task.

Often the statements change the state of the program, like updating global variables. A classic example is writing a for loop that gives exact directions to iterate over the indices of an array.

In contrast, functional programming is a form of declarative programming. You tell the computer what you want done by calling a method or function.

JavaScript offers many predefined methods that handle common tasks so you don't need to write out how the computer should perform them. For example, instead of using the for loop mentioned above, you could call the map method which handles the details of iterating over an array. This helps to avoid semantic errors, like the "Off By One Errors" that were covered in the Debugging section.

Consider the scenario: you are browsing the web in your browser, and want to track the tabs you have opened. Let's try to model this using some simple object-oriented code.

A Window object is made up of tabs, and you usually have more than one Window open. The titles of each open site in each Window object is held in an array. After working in the browser (opening new tabs, merging windows, and closing tabs), you want to print the tabs that are still open. Closed tabs are removed from the array and new tabs (for simplicity) get added to the end of it.

The code editor shows an implementation of this functionality with functions for tabOpen(), tabClose(), and join(). The array tabs is part of the Window object that stores the name of the open pages.

Examine the code in the editor. It's using a method that has side effects in the program, causing incorrect behaviour. The final list of open tabs, stored in finalTabs.tabs, should be ['FB', 'Gitter', 'Reddit', 'Twitter', 'Medium', 'new tab', 'Netflix', 'YouTube', 'Vine', 'GMail', 'Work mail', 'Docs', 'freeCodeCamp', 'new tab'] but the list produced by the code is slightly different.

Change Window.prototype.tabClose so that it removes the correct tab.

// tabs is an array of titles of each site open within the window

var Window = function(tabs) {

this.tabs = tabs; // We keep a record of the array inside the object

};

// When you join two windows into one window

Window.prototype.join = function (otherWindow) {

this.tabs = this.tabs.concat(otherWindow.tabs);

return this;

};

// When you open a new tab at the end

Window.prototype.tabOpen = function (tab) {

this.tabs.push('new tab'); // Let's open a new tab for now

return this;

};

// When you close a tab

Window.prototype.tabClose = function (index) {

// Only change code below this line

var tabsBeforeIndex = this.tabs.slice(0, index); // Get the tabs before the tab

var tabsAfterIndex = this.tabs.slice(index + 1); // Get the tabs after the tab

this.tabs = tabsBeforeIndex.concat(tabsAfterIndex); // Join them together

// Only change code above this line

return this;

};

// Let's create three browser windows

var workWindow = new Window(['GMail', 'Inbox', 'Work mail', 'Docs', 'freeCodeCamp']); // Your mailbox, drive, and other work sites

var socialWindow = new Window(['FB', 'Gitter', 'Reddit', 'Twitter', 'Medium']); // Social sites

var videoWindow = new Window(['Netflix', 'YouTube', 'Vimeo', 'Vine']); // Entertainment sites

// Now perform the tab opening, closing, and other operations

var finalTabs = socialWindow

.tabOpen() // Open a new tab for cat memes

.join(videoWindow.tabClose(2)) // Close third tab in video window, and join

.join(workWindow.tabClose(1).tabOpen());

console.log(finalTabs.tabs);

\*\*\*\*\*\*

**Avoid Mutations and Side Effects Using Functional Programming**

If you haven't already figured it out, the issue in the previous challenge was with the splice call in the tabClose() function. Unfortunately, splice changes the original array it is called on, so the second call to it used a modified array, and gave unexpected results.

This is a small example of a much larger pattern - you call a function on a variable, array, or an object, and the function changes the variable or something in the object.

One of the core principles of functional programming is to not change things. Changes lead to bugs. It's easier to prevent bugs knowing that your functions don't change anything, including the function arguments or any global variable.

The previous example didn't have any complicated operations but the splice method changed the original array, and resulted in a bug.

Recall that in functional programming, changing or altering things is called mutation, and the outcome is called a side effect. A function, ideally, should be a pure function, meaning that it does not cause any side effects.

Let's try to master this discipline and not alter any variable or object in our code.

Fill in the code for the function incrementer so it returns the value of the global variable fixedValue increased by one.

// The global variable

var fixedValue = 4;

function incrementer () {

// Only change code below this line

return fixedValue + 1;

// Only change code above this line

}

var newValue = incrementer(); // Should equal 5

console.log(fixedValue); // Should print 4

\*\*\*\*\*

**Pass Arguments to Avoid External Dependence in a Function**

The last challenge was a step closer to functional programming principles, but there is still something missing.

We didn't alter the global variable value, but the function incrementer would not work without the global variable fixedValue being there.

Another principle of functional programming is to always declare your dependencies explicitly. This means if a function depends on a variable or object being present, then pass that variable or object directly into the function as an argument.

There are several good consequences from this principle. The function is easier to test, you know exactly what input it takes, and it won't depend on anything else in your program.

This can give you more confidence when you alter, remove, or add new code. You would know what you can or cannot change and you can see where the potential traps are.

Finally, the function would always produce the same output for the same set of inputs, no matter what part of the code executes it.

Let's update the incrementer function to clearly declare its dependencies.

Write the incrementer function so it takes an argument, and then returns a result after increasing the value by one.

// The global variable

var fixedValue = 4;

// Only change code below this line

function incrementer (obj) {

return obj+1;

// Only change code above this line

}

\*\*\*\*\*\*\*

**Refactor Global Variables Out of Functions**

So far, we have seen two distinct principles for functional programming:

1. Don't alter a variable or object - create new variables and objects and return them if need be from a function. Hint: using something like var newArr = arrVar, where arrVar is an array will simply create a reference to the existing variable and not a copy. So changing a value in newArr would change the value in arrVar.
2. Declare function parameters - any computation inside a function depends only on the arguments passed to the function, and not on any global object or variable.

Adding one to a number is not very exciting, but we can apply these principles when working with arrays or more complex objects.

Rewrite the code so the global array bookList is not changed inside either function. The add function should add the given bookName to the end of the array passed to it and return a new array (list). The remove function should remove the given bookName from the array passed to it.

**Note:** Both functions should return an array, and any new parameters should be added before the bookName parameter.

var bookList = ["The Hound of the Baskervilles", "On The Electrodynamics of Moving Bodies", "Philosophiæ Naturalis Principia Mathematica", "Disquisitiones Arithmeticae"];

//function add a book to the array

function add( bookList, bookName ) {

//make a copy of the argument's array

let bookListCopy = bookList.slice(0);

//add the bookName to the new array

bookListCopy.push( bookName );

//return the new modified array

return bookListCopy;

};

//function remove a book from the array

function remove( bookList, bookName ) {

//make a copy of the argument's array

let bookListCopy = bookList.slice(0);

if( bookListCopy.indexOf( bookName ) >= 0 ) {

//delet the book

bookListCopy.splice( bookListCopy.indexOf( bookName ), 1 );

//return the new modified array

return bookListCopy;

}

};

//create a new array from an already existing one but with a new bookName in it

var newBookList = add( bookList, 'A Brief History of Time' );

//create a new array from an already existing one but with a deleted bookName from it

var newerBookList = remove( bookList, 'On The Electrodynamics of Moving Bodies' );

//remove from the bookList at which we added 'A Brief History of Time', the bookName 'On The Electrodynamics of Moving Bodies'

var newestBookList = remove( newBookList, 'On The Electrodynamics of Moving Bodies' );

//["The Hound of the Baskervilles", "On The Electrodynamics of Moving Bodies", "Philosophiæ Naturalis Principia Mathematica", "Disquisitiones Arithmeticae"];

console.log( bookList );

//["The Hound of the Baskervilles", "Philosophiæ Naturalis Principia Mathematica", "Disquisitiones Arithmeticae"]

console.log( newerBookList );

//["The Hound of the Baskervilles", "Philosophiæ Naturalis Principia Mathematica", "Disquisitiones Arithmeticae", "A Brief History of Time"]

console.log( newestBookList );

Second solution:

// The global variable

var bookList = ["The Hound of the Baskervilles", "On The Electrodynamics of Moving Bodies", "Philosophiæ Naturalis Principia Mathematica", "Disquisitiones Arithmeticae"];

// Change code below this line

function add (list, bookName) {

//use the spread operator to make a copy of the bookList array and to add the given bookName to this new array

return [...list, bookName];

// Add your code above this line

}

/\* This function should remove a book from the list and return the list \*/

// New parameters should come before the bookName one

// Add your code below this line

function remove (list, bookName) {

if (list.indexOf(bookName) >= 0) {

//use filter to remove bookName from the bookList array

//filter will return a new array with all elements or items that pass the test implemented by the provided function

//in this case, filter will return all items from the array that are not equal to bookName

//so bookName will be filtered out of this new array

return list.filter((item) => item !== bookName);

// Add your code above this line

}

}

var newBookList = add(bookList, 'A Brief History of Time');

var newerBookList = remove(bookList, 'On The Electrodynamics of Moving Bodies');

var newestBookList = remove(add(bookList, 'A Brief History of Time'), 'On The Electrodynamics of Moving Bodies');

console.log(bookList);

Final solution:

// The global variable

var bookList = ["The Hound of the Baskervilles", "On The Electrodynamics of Moving Bodies", "Philosophiæ Naturalis Principia Mathematica", "Disquisitiones Arithmeticae"];

// Change code below this line

function add (list, bookName) {

return [...list, bookName];

// Add your code above this line

}

function remove (list, bookName) {

if (list.indexOf(bookName) >= 0) {

return list.filter((item) => item !== bookName);

// Add your code above this line

}

}

var newBookList = add(bookList, 'A Brief History of Time');

var newerBookList = remove(bookList, 'On The Electrodynamics of Moving Bodies');

var newestBookList = remove(add(bookList, 'A Brief History of Time'), 'On The Electrodynamics of Moving Bodies');

console.log(bookList);

\*\*\*\*\*\*

**Use the map Method to Extract Data from an Array**

So far we have learned to use pure functions to avoid side effects in a program. Also, we have seen the value in having a function only depend on its input arguments.

This is only the beginning. As its name suggests, functional programming is centered around a theory of functions.

It would make sense to be able to pass them as arguments to other functions, and return a function from another function. Functions are considered first class objects in JavaScript, which means they can be used like any other object. They can be saved in variables, stored in an object, or passed as function arguments.

Let's start with some simple array functions, which are methods on the array object prototype. In this exercise we are looking at Array.prototype.map(), or more simply map.

The map method iterates over each item in an array and returns a new array containing the results of calling the callback function on each element. It does this without mutating the original array.

When the callback is used, it is passed three arguments. The first argument is the current element being processed. The second is the index of that element and the third is the array upon which the map method was called.

See below for an example using the map method on the users array to return a new array containing only the names of the users as elements. For simplicity, the example only uses the first argument of the callback.

const users = [

{ name: 'John', age: 34 },

{ name: 'Amy', age: 20 },

{ name: 'camperCat', age: 10 }

];

const names = users.map(user => user.name);

console.log(names);

The console would display the value [ 'John', 'Amy', 'camperCat' ].

The watchList array holds objects with information on several movies. Use map on watchList to assign a new array of objects with only title and rating keys to the ratings variable. The code in the editor currently uses a for loop to do this, so you should replace the loop functionality with your map expression.

// The global variable

var watchList = [

{

"Title": "Inception",

"Year": "2010",

"Rated": "PG-13",

"Released": "16 Jul 2010",

"Runtime": "148 min",

"Genre": "Action, Adventure, Crime",

"Director": "Christopher Nolan",

"Writer": "Christopher Nolan",

"Actors": "Leonardo DiCaprio, Joseph Gordon-Levitt, Elliot Page, Tom Hardy",

"Plot": "A thief, who steals corporate secrets through use of dream-sharing technology, is given the inverse task of planting an idea into the mind of a CEO.",

"Language": "English, Japanese, French",

"Country": "USA, UK",

"Awards": "Won 4 Oscars. Another 143 wins & 198 nominations.",

"Poster": "http://ia.media-imdb.com/images/M/MV5BMjAxMzY3NjcxNF5BMl5BanBnXkFtZTcwNTI5OTM0Mw@@.\_V1\_SX300.jpg",

"Metascore": "74",

"imdbRating": "8.8",

"imdbVotes": "1,446,708",

"imdbID": "tt1375666",

"Type": "movie",

"Response": "True"

},

{

"Title": "Interstellar",

"Year": "2014",

"Rated": "PG-13",

"Released": "07 Nov 2014",

"Runtime": "169 min",

"Genre": "Adventure, Drama, Sci-Fi",

"Director": "Christopher Nolan",

"Writer": "Jonathan Nolan, Christopher Nolan",

"Actors": "Ellen Burstyn, Matthew McConaughey, Mackenzie Foy, John Lithgow",

"Plot": "A team of explorers travel through a wormhole in space in an attempt to ensure humanity's survival.",

"Language": "English",

"Country": "USA, UK",

"Awards": "Won 1 Oscar. Another 39 wins & 132 nominations.",

"Poster": "http://ia.media-imdb.com/images/M/MV5BMjIxNTU4MzY4MF5BMl5BanBnXkFtZTgwMzM4ODI3MjE@.\_V1\_SX300.jpg",

"Metascore": "74",

"imdbRating": "8.6",

"imdbVotes": "910,366",

"imdbID": "tt0816692",

"Type": "movie",

"Response": "True"

},

{

"Title": "The Dark Knight",

"Year": "2008",

"Rated": "PG-13",

"Released": "18 Jul 2008",

"Runtime": "152 min",

"Genre": "Action, Adventure, Crime",

"Director": "Christopher Nolan",

"Writer": "Jonathan Nolan (screenplay), Christopher Nolan (screenplay), Christopher Nolan (story), David S. Goyer (story), Bob Kane (characters)",

"Actors": "Christian Bale, Heath Ledger, Aaron Eckhart, Michael Caine",

"Plot": "When the menace known as the Joker wreaks havoc and chaos on the people of Gotham, the caped crusader must come to terms with one of the greatest psychological tests of his ability to fight injustice.",

"Language": "English, Mandarin",

"Country": "USA, UK",

"Awards": "Won 2 Oscars. Another 146 wins & 142 nominations.",

"Poster": "http://ia.media-imdb.com/images/M/MV5BMTMxNTMwODM0NF5BMl5BanBnXkFtZTcwODAyMTk2Mw@@.\_V1\_SX300.jpg",

"Metascore": "82",

"imdbRating": "9.0",

"imdbVotes": "1,652,832",

"imdbID": "tt0468569",

"Type": "movie",

"Response": "True"

},

{

"Title": "Batman Begins",

"Year": "2005",

"Rated": "PG-13",

"Released": "15 Jun 2005",

"Runtime": "140 min",

"Genre": "Action, Adventure",

"Director": "Christopher Nolan",

"Writer": "Bob Kane (characters), David S. Goyer (story), Christopher Nolan (screenplay), David S. Goyer (screenplay)",

"Actors": "Christian Bale, Michael Caine, Liam Neeson, Katie Holmes",

"Plot": "After training with his mentor, Batman begins his fight to free crime-ridden Gotham City from the corruption that Scarecrow and the League of Shadows have cast upon it.",

"Language": "English, Urdu, Mandarin",

"Country": "USA, UK",

"Awards": "Nominated for 1 Oscar. Another 15 wins & 66 nominations.",

"Poster": "http://ia.media-imdb.com/images/M/MV5BNTM3OTc0MzM2OV5BMl5BanBnXkFtZTYwNzUwMTI3.\_V1\_SX300.jpg",

"Metascore": "70",

"imdbRating": "8.3",

"imdbVotes": "972,584",

"imdbID": "tt0372784",

"Type": "movie",

"Response": "True"

},

{

"Title": "Avatar",

"Year": "2009",

"Rated": "PG-13",

"Released": "18 Dec 2009",

"Runtime": "162 min",

"Genre": "Action, Adventure, Fantasy",

"Director": "James Cameron",

"Writer": "James Cameron",

"Actors": "Sam Worthington, Zoe Saldana, Sigourney Weaver, Stephen Lang",

"Plot": "A paraplegic marine dispatched to the moon Pandora on a unique mission becomes torn between following his orders and protecting the world he feels is his home.",

"Language": "English, Spanish",

"Country": "USA, UK",

"Awards": "Won 3 Oscars. Another 80 wins & 121 nominations.",

"Poster": "http://ia.media-imdb.com/images/M/MV5BMTYwOTEwNjAzMl5BMl5BanBnXkFtZTcwODc5MTUwMw@@.\_V1\_SX300.jpg",

"Metascore": "83",

"imdbRating": "7.9",

"imdbVotes": "876,575",

"imdbID": "tt0499549",

"Type": "movie",

"Response": "True"

}

];

var ratings = watchList.map(function(movie) {

return {

title: movie["Title"],

rating: movie["imdbRating"]

}

});

\*\*\*\*\*\*\*

**Implement map on a Prototype**

As you have seen from applying Array.prototype.map(), or simply map() earlier, the map method returns an array of the same length as the one it was called on. It also doesn't alter the original array, as long as its callback function doesn't.

In other words, map is a pure function, and its output depends solely on its inputs. Plus, it takes another function as its argument.

You might learn a lot about the map method if you implement your own version of it. It is recommended you use a for loop or Array.prototype.forEach().

Write your own Array.prototype.myMap(), which should behave exactly like Array.prototype.map(). You should not use the built-in map method. The Array instance can be accessed in the myMap method using this.

// The global variable

var s = [23, 65, 98, 5];

Array.prototype.myMap = function(callback) {

var newArray = [];

// Only change code below this line

this.forEach(a=>newArray.push(callback(a)));

// Only change code above this line

return newArray;

};

var new\_s = s.myMap(function(item) {

return item \* 2;

});

\*\*\*\*\*\*\*

**Use the filter Method to Extract Data from an Array**

Another useful array function is Array.prototype.filter(), or simply filter().

filter calls a function on each element of an array and returns a new array containing only the elements for which that function returns true. In other words, it filters the array, based on the function passed to it. Like map, it does this without needing to modify the original array.

The callback function accepts three arguments. The first argument is the current element being processed. The second is the index of that element and the third is the array upon which the filter method was called.

See below for an example using the filter method on the users array to return a new array containing only the users under the age of 30. For simplicity, the example only uses the first argument of the callback.

const users = [

{ name: 'John', age: 34 },

{ name: 'Amy', age: 20 },

{ name: 'camperCat', age: 10 }

];

const usersUnder30 = users.filter(user => user.age < 30);

console.log(usersUnder30);

The console would display the value [ { name: 'Amy', age: 20 }, { name: 'camperCat', age: 10 } ].

The variable watchList holds an array of objects with information on several movies. Use a combination of filter and map on watchList to assign a new array of objects with only title and rating keys. The new array should only include objects where imdbRating is greater than or equal to 8.0. Note that the rating values are saved as strings in the object and you may need to convert them into numbers to perform mathematical operations on them.

// The global variable

var watchList = [

{

"Title": "Inception",

"Year": "2010",

"Rated": "PG-13",

"Released": "16 Jul 2010",

"Runtime": "148 min",

"Genre": "Action, Adventure, Crime",

"Director": "Christopher Nolan",

"Writer": "Christopher Nolan",

"Actors": "Leonardo DiCaprio, Joseph Gordon-Levitt, Elliot Page, Tom Hardy",

"Plot": "A thief, who steals corporate secrets through use of dream-sharing technology, is given the inverse task of planting an idea into the mind of a CEO.",

"Language": "English, Japanese, French",

"Country": "USA, UK",

"Awards": "Won 4 Oscars. Another 143 wins & 198 nominations.",

"Poster": "http://ia.media-imdb.com/images/M/MV5BMjAxMzY3NjcxNF5BMl5BanBnXkFtZTcwNTI5OTM0Mw@@.\_V1\_SX300.jpg",

"Metascore": "74",

"imdbRating": "8.8",

"imdbVotes": "1,446,708",

"imdbID": "tt1375666",

"Type": "movie",

"Response": "True"

},

{

"Title": "Interstellar",

"Year": "2014",

"Rated": "PG-13",

"Released": "07 Nov 2014",

"Runtime": "169 min",

"Genre": "Adventure, Drama, Sci-Fi",

"Director": "Christopher Nolan",

"Writer": "Jonathan Nolan, Christopher Nolan",

"Actors": "Ellen Burstyn, Matthew McConaughey, Mackenzie Foy, John Lithgow",

"Plot": "A team of explorers travel through a wormhole in space in an attempt to ensure humanity's survival.",

"Language": "English",

"Country": "USA, UK",

"Awards": "Won 1 Oscar. Another 39 wins & 132 nominations.",

"Poster": "http://ia.media-imdb.com/images/M/MV5BMjIxNTU4MzY4MF5BMl5BanBnXkFtZTgwMzM4ODI3MjE@.\_V1\_SX300.jpg",

"Metascore": "74",

"imdbRating": "8.6",

"imdbVotes": "910,366",

"imdbID": "tt0816692",

"Type": "movie",

"Response": "True"

},

{

"Title": "The Dark Knight",

"Year": "2008",

"Rated": "PG-13",

"Released": "18 Jul 2008",

"Runtime": "152 min",

"Genre": "Action, Adventure, Crime",

"Director": "Christopher Nolan",

"Writer": "Jonathan Nolan (screenplay), Christopher Nolan (screenplay), Christopher Nolan (story), David S. Goyer (story), Bob Kane (characters)",

"Actors": "Christian Bale, Heath Ledger, Aaron Eckhart, Michael Caine",

"Plot": "When the menace known as the Joker wreaks havoc and chaos on the people of Gotham, the caped crusader must come to terms with one of the greatest psychological tests of his ability to fight injustice.",

"Language": "English, Mandarin",

"Country": "USA, UK",

"Awards": "Won 2 Oscars. Another 146 wins & 142 nominations.",

"Poster": "http://ia.media-imdb.com/images/M/MV5BMTMxNTMwODM0NF5BMl5BanBnXkFtZTcwODAyMTk2Mw@@.\_V1\_SX300.jpg",

"Metascore": "82",

"imdbRating": "9.0",

"imdbVotes": "1,652,832",

"imdbID": "tt0468569",

"Type": "movie",

"Response": "True"

},

{

"Title": "Batman Begins",

"Year": "2005",

"Rated": "PG-13",

"Released": "15 Jun 2005",

"Runtime": "140 min",

"Genre": "Action, Adventure",

"Director": "Christopher Nolan",

"Writer": "Bob Kane (characters), David S. Goyer (story), Christopher Nolan (screenplay), David S. Goyer (screenplay)",

"Actors": "Christian Bale, Michael Caine, Liam Neeson, Katie Holmes",

"Plot": "After training with his mentor, Batman begins his fight to free crime-ridden Gotham City from the corruption that Scarecrow and the League of Shadows have cast upon it.",

"Language": "English, Urdu, Mandarin",

"Country": "USA, UK",

"Awards": "Nominated for 1 Oscar. Another 15 wins & 66 nominations.",

"Poster": "http://ia.media-imdb.com/images/M/MV5BNTM3OTc0MzM2OV5BMl5BanBnXkFtZTYwNzUwMTI3.\_V1\_SX300.jpg",

"Metascore": "70",

"imdbRating": "8.3",

"imdbVotes": "972,584",

"imdbID": "tt0372784",

"Type": "movie",

"Response": "True"

},

{

"Title": "Avatar",

"Year": "2009",

"Rated": "PG-13",

"Released": "18 Dec 2009",

"Runtime": "162 min",

"Genre": "Action, Adventure, Fantasy",

"Director": "James Cameron",

"Writer": "James Cameron",

"Actors": "Sam Worthington, Zoe Saldana, Sigourney Weaver, Stephen Lang",

"Plot": "A paraplegic marine dispatched to the moon Pandora on a unique mission becomes torn between following his orders and protecting the world he feels is his home.",

"Language": "English, Spanish",

"Country": "USA, UK",

"Awards": "Won 3 Oscars. Another 80 wins & 121 nominations.",

"Poster": "http://ia.media-imdb.com/images/M/MV5BMTYwOTEwNjAzMl5BMl5BanBnXkFtZTcwODc5MTUwMw@@.\_V1\_SX300.jpg",

"Metascore": "83",

"imdbRating": "7.9",

"imdbVotes": "876,575",

"imdbID": "tt0499549",

"Type": "movie",

"Response": "True"

}

];

// Only change code below this line

var filteredList = [];

var filtered = [];

filtered = watchList.filter( (item) => item["imdbRating"] >= 8 );

filteredList = filtered.map( (item) => ({"title": item["Title"], "rating":item["imdbRating"]}) );

// Only change code above this line

console.log(filteredList);

\*\*\*\*\*\*

**Implement the filter Method on a Prototype**

You might learn a lot about the filter method if you implement your own version of it. It is recommended you use a for loop or Array.prototype.forEach().

Write your own Array.prototype.myFilter(), which should behave exactly like Array.prototype.filter(). You should not use the built-in filter method. The Array instance can be accessed in the myFilter method using this.

// The global variable

var s = [23, 65, 98, 5];

Array.prototype.myFilter = function(callback) {

// Only change code below this line

var newArray = [];

this.forEach(function(x){

if(callback(x)===true){

newArray.push(x);

}

})

// Only change code above this line

return newArray;

};

var new\_s = s.myFilter(function(item) {

return item % 2 === 1;

});

\*\*\*\*\*\*

**Return Part of an Array Using the slice Method**

The slice method returns a copy of certain elements of an array. It can take two arguments, the first gives the index of where to begin the slice, the second is the index for where to end the slice (and it's non-inclusive). If the arguments are not provided, the default is to start at the beginning of the array through the end, which is an easy way to make a copy of the entire array. The slice method does not mutate the original array, but returns a new one.

Here's an example:

var arr = ["Cat", "Dog", "Tiger", "Zebra"];

var newArray = arr.slice(1, 3);

newArray would have the value ["Dog", "Tiger"].

Use the slice method in the sliceArray function to return part of the anim array given the provided beginSlice and endSlice indices. The function should return an array.

function sliceArray(anim, beginSlice, endSlice) {

// Only change code below this line

var result=anim.slice(beginSlice,endSlice);

return result;

// Only change code above this line

}

var inputAnim = ["Cat", "Dog", "Tiger", "Zebra", "Ant"];

console.log(sliceArray(inputAnim, 1, 3));

// running tests

// tests completed

// console output

[ 'Dog', 'Tiger' ]

[ 'Dog', 'Tiger' ]

Another way by just not creating variable just add return statement:

function sliceArray(anim, beginSlice, endSlice) {

// Only change code below this line

return anim.slice(beginSlice,endSlice);

// Only change code above this line

}

var inputAnim = ["Cat", "Dog", "Tiger", "Zebra", "Ant"];

console.log(sliceArray(inputAnim, 1, 3));

\*\*\*\*\*

**Remove Elements from an Array Using slice Instead of splice**

A common pattern while working with arrays is when you want to remove items and keep the rest of the array. JavaScript offers the splice method for this, which takes arguments for the index of where to start removing items, then the number of items to remove. If the second argument is not provided, the default is to remove items through the end. However, the splice method mutates the original array it is called on. Here's an example:

var cities = ["Chicago", "Delhi", "Islamabad", "London", "Berlin"];

cities.splice(3, 1);

Here splice returns the string London and deletes it from the cities array. cities will have the value ["Chicago", "Delhi", "Islamabad", "Berlin"].

As we saw in the last challenge, the slice method does not mutate the original array, but returns a new one which can be saved into a variable. Recall that the slice method takes two arguments for the indices to begin and end the slice (the end is non-inclusive), and returns those items in a new array. Using the slice method instead of splice helps to avoid any array-mutating side effects.

Rewrite the function nonMutatingSplice by using slice instead of splice. It should limit the provided cities array to a length of 3, and return a new array with only the first three items.

Do not mutate the original array provided to the function.

function nonMutatingSplice(cities) {

// Only change code below this line

return cities.slice(0,3);

// Only change code above this line

}

var inputCities = ["Chicago", "Delhi", "Islamabad", "London", "Berlin"];

nonMutatingSplice(inputCities);

\*\*\*\*\*\*

**Combine Two Arrays Using the concat Method**

Concatenation means to join items end to end. JavaScript offers the concat method for both strings and arrays that work in the same way. For arrays, the method is called on one, then another array is provided as the argument to concat, which is added to the end of the first array. It returns a new array and does not mutate either of the original arrays. Here's an example:

[1, 2, 3].concat([4, 5, 6]);

The returned array would be [1, 2, 3, 4, 5, 6].

Use the concat method in the nonMutatingConcat function to concatenate attach to the end of original. The function should return the concatenated array.

function nonMutatingConcat(original, attach) {

// Only change code below this line

return original.concat(attach);

// Only change code above this line

}

var first = [1, 2, 3];

var second = [4, 5];

nonMutatingConcat(first, second);

\*\*\*\*\*

**Add Elements to the End of an Array Using concat Instead of push**

Functional programming is all about creating and using non-mutating functions.

The last challenge introduced the concat method as a way to combine arrays into a new one without mutating the original arrays. Compare concat to the push method. push adds an item to the end of the same array it is called on, which mutates that array. Here's an example:

var arr = [1, 2, 3];

arr.push([4, 5, 6]);

arr would have a modified value of [1, 2, 3, [4, 5, 6]], which is not the functional programming way.

concat offers a way to add new items to the end of an array without any mutating side effects.

Change the nonMutatingPush function so it uses concat to add newItem to the end of original instead of push. The function should return an array.

function nonMutatingPush(original, newItem) {

// Only change code below this line

return original.concat(newItem);

// Only change code above this line

}

var first = [1, 2, 3];

var second = [4, 5];

nonMutatingPush(first, second);

\*\*\*\*\*\*

**Use the reduce Method to Analyze Data**

Array.prototype.reduce(), or simply reduce(), is the most general of all array operations in JavaScript. You can solve almost any array processing problem using the reduce method.

The reduce method allows for more general forms of array processing, and it's possible to show that both filter and map can be derived as special applications of reduce. The reduce method iterates over each item in an array and returns a single value (i.e. string, number, object, array). This is achieved via a callback function that is called on each iteration.

The callback function accepts four arguments. The first argument is known as the accumulator, which gets assigned the return value of the callback function from the previous iteration, the second is the current element being processed, the third is the index of that element and the fourth is the array upon which reduce is called.

In addition to the callback function, reduce has an additional parameter which takes an initial value for the accumulator. If this second parameter is not used, then the first iteration is skipped and the second iteration gets passed the first element of the array as the accumulator.

See below for an example using reduce on the users array to return the sum of all the users' ages. For simplicity, the example only uses the first and second arguments.

const users = [

{ name: 'John', age: 34 },

{ name: 'Amy', age: 20 },

{ name: 'camperCat', age: 10 }

];

const sumOfAges = users.reduce((sum, user) => sum + user.age, 0);

console.log(sumOfAges);

The console would display the value 64.

In another example, see how an object can be returned containing the names of the users as properties with their ages as values.

const users = [

{ name: 'John', age: 34 },

{ name: 'Amy', age: 20 },

{ name: 'camperCat', age: 10 }

];

const usersObj = users.reduce((obj, user) => {

obj[user.name] = user.age;

return obj;

}, {});

console.log(usersObj);

The console would display the value { John: 34, Amy: 20, camperCat: 10 }.

The variable watchList holds an array of objects with information on several movies. Use reduce to find the average IMDB rating of the movies directed by Christopher Nolan. Recall from prior challenges how to filter data and map over it to pull what you need. You may need to create other variables, and return the average rating from getRating function. Note that the rating values are saved as strings in the object and need to be converted into numbers before they are used in any mathematical operations.

// The global variable

var watchList = [

{

"Title": "Inception",

"Year": "2010",

"Rated": "PG-13",

"Released": "16 Jul 2010",

"Runtime": "148 min",

"Genre": "Action, Adventure, Crime",

"Director": "Christopher Nolan",

"Writer": "Christopher Nolan",

"Actors": "Leonardo DiCaprio, Joseph Gordon-Levitt, Elliot Page, Tom Hardy",

"Plot": "A thief, who steals corporate secrets through use of dream-sharing technology, is given the inverse task of planting an idea into the mind of a CEO.",

"Language": "English, Japanese, French",

"Country": "USA, UK",

"Awards": "Won 4 Oscars. Another 143 wins & 198 nominations.",

"Poster": "http://ia.media-imdb.com/images/M/MV5BMjAxMzY3NjcxNF5BMl5BanBnXkFtZTcwNTI5OTM0Mw@@.\_V1\_SX300.jpg",

"Metascore": "74",

"imdbRating": "8.8",

"imdbVotes": "1,446,708",

"imdbID": "tt1375666",

"Type": "movie",

"Response": "True"

},

{

"Title": "Interstellar",

"Year": "2014",

"Rated": "PG-13",

"Released": "07 Nov 2014",

"Runtime": "169 min",

"Genre": "Adventure, Drama, Sci-Fi",

"Director": "Christopher Nolan",

"Writer": "Jonathan Nolan, Christopher Nolan",

"Actors": "Ellen Burstyn, Matthew McConaughey, Mackenzie Foy, John Lithgow",

"Plot": "A team of explorers travel through a wormhole in space in an attempt to ensure humanity's survival.",

"Language": "English",

"Country": "USA, UK",

"Awards": "Won 1 Oscar. Another 39 wins & 132 nominations.",

"Poster": "http://ia.media-imdb.com/images/M/MV5BMjIxNTU4MzY4MF5BMl5BanBnXkFtZTgwMzM4ODI3MjE@.\_V1\_SX300.jpg",

"Metascore": "74",

"imdbRating": "8.6",

"imdbVotes": "910,366",

"imdbID": "tt0816692",

"Type": "movie",

"Response": "True"

},

{

"Title": "The Dark Knight",

"Year": "2008",

"Rated": "PG-13",

"Released": "18 Jul 2008",

"Runtime": "152 min",

"Genre": "Action, Adventure, Crime",

"Director": "Christopher Nolan",

"Writer": "Jonathan Nolan (screenplay), Christopher Nolan (screenplay), Christopher Nolan (story), David S. Goyer (story), Bob Kane (characters)",

"Actors": "Christian Bale, Heath Ledger, Aaron Eckhart, Michael Caine",

"Plot": "When the menace known as the Joker wreaks havoc and chaos on the people of Gotham, the caped crusader must come to terms with one of the greatest psychological tests of his ability to fight injustice.",

"Language": "English, Mandarin",

"Country": "USA, UK",

"Awards": "Won 2 Oscars. Another 146 wins & 142 nominations.",

"Poster": "http://ia.media-imdb.com/images/M/MV5BMTMxNTMwODM0NF5BMl5BanBnXkFtZTcwODAyMTk2Mw@@.\_V1\_SX300.jpg",

"Metascore": "82",

"imdbRating": "9.0",

"imdbVotes": "1,652,832",

"imdbID": "tt0468569",

"Type": "movie",

"Response": "True"

},

{

"Title": "Batman Begins",

"Year": "2005",

"Rated": "PG-13",

"Released": "15 Jun 2005",

"Runtime": "140 min",

"Genre": "Action, Adventure",

"Director": "Christopher Nolan",

"Writer": "Bob Kane (characters), David S. Goyer (story), Christopher Nolan (screenplay), David S. Goyer (screenplay)",

"Actors": "Christian Bale, Michael Caine, Liam Neeson, Katie Holmes",

"Plot": "After training with his mentor, Batman begins his fight to free crime-ridden Gotham City from the corruption that Scarecrow and the League of Shadows have cast upon it.",

"Language": "English, Urdu, Mandarin",

"Country": "USA, UK",

"Awards": "Nominated for 1 Oscar. Another 15 wins & 66 nominations.",

"Poster": "http://ia.media-imdb.com/images/M/MV5BNTM3OTc0MzM2OV5BMl5BanBnXkFtZTYwNzUwMTI3.\_V1\_SX300.jpg",

"Metascore": "70",

"imdbRating": "8.3",

"imdbVotes": "972,584",

"imdbID": "tt0372784",

"Type": "movie",

"Response": "True"

},

{

"Title": "Avatar",

"Year": "2009",

"Rated": "PG-13",

"Released": "18 Dec 2009",

"Runtime": "162 min",

"Genre": "Action, Adventure, Fantasy",

"Director": "James Cameron",

"Writer": "James Cameron",

"Actors": "Sam Worthington, Zoe Saldana, Sigourney Weaver, Stephen Lang",

"Plot": "A paraplegic marine dispatched to the moon Pandora on a unique mission becomes torn between following his orders and protecting the world he feels is his home.",

"Language": "English, Spanish",

"Country": "USA, UK",

"Awards": "Won 3 Oscars. Another 80 wins & 121 nominations.",

"Poster": "http://ia.media-imdb.com/images/M/MV5BMTYwOTEwNjAzMl5BMl5BanBnXkFtZTcwODc5MTUwMw@@.\_V1\_SX300.jpg",

"Metascore": "83",

"imdbRating": "7.9",

"imdbVotes": "876,575",

"imdbID": "tt0499549",

"Type": "movie",

"Response": "True"

}

];

function getRating(watchList){

// Only change code below this line

const averageRating = watchList.filter(x => x.Director === "Christopher Nolan").map(x => Number(x.imdbRating)).reduce((x1, x2) => x1 + x2) / watchList.filter(x => x.Director === "Christopher Nolan").length;

// Only change code above this line

return averageRating;

}

console.log(getRating(watchList));

\*\*\*\*\*\*\*

**Use Higher-Order Functions map, filter, or reduce to Solve a Complex Problem**

Now that you have worked through a few challenges using higher-order functions like map(), filter(), and reduce(), you now get to apply them to solve a more complex challenge.

Complete the code for the squareList function using any combination of map(), filter(), and reduce(). The function should return a new array containing the squares of *only* the positive integers (decimal numbers are not integers) when an array of real numbers is passed to it. An example of an array of real numbers is [-3, 4.8, 5, 3, -3.2].

**Note:** Your function should not use any kind of for or while loops or the forEach() function.

const squareList = arr => {

// Only change code below this line

const positiveinteger=arr.filter(num=>{

return num>=0 &&Number.isInteger(num);

});

const squaredIntegers=positiveinteger.map(num=>{

return num \*\*2;

})

return squaredIntegers;

// Only change code above this line

};

const squaredIntegers = squareList([-3, 4.8, 5, 3, -3.2]);

console.log(squaredIntegers);

\*\*\*\*\*\*\*

**Sort an Array Alphabetically using the sort Method**

The sort method sorts the elements of an array according to the callback function.

For example:

function ascendingOrder(arr) {

return arr.sort(function(a, b) {

return a - b;

});

}

ascendingOrder([1, 5, 2, 3, 4]);

This would return the value [1, 2, 3, 4, 5].

function reverseAlpha(arr) {

return arr.sort(function(a, b) {

return a === b ? 0 : a < b ? 1 : -1;

});

}

reverseAlpha(['l', 'h', 'z', 'b', 's']);

This would return the value ['z', 's', 'l', 'h', 'b'].

JavaScript's default sorting method is by string Unicode point value, which may return unexpected results. Therefore, it is encouraged to provide a callback function to specify how to sort the array items. When such a callback function, normally called compareFunction, is supplied, the array elements are sorted according to the return value of the compareFunction: If compareFunction(a,b) returns a value less than 0 for two elements a and b, then a will come before b. If compareFunction(a,b) returns a value greater than 0 for two elements a and b, then b will come before a. If compareFunction(a,b) returns a value equal to 0 for two elements a and b, then a and b will remain unchanged.

Use the sort method in the alphabeticalOrder function to sort the elements of arr in alphabetical order.

function alphabeticalOrder(arr) {

// Only change code below this line

return arr.sort();

// Only change code above this line

}

alphabeticalOrder(["a", "d", "c", "a", "z", "g"]);

\*\*\*\*\*\*\*

**Return a Sorted Array Without Changing the Original Array**

A side effect of the sort method is that it changes the order of the elements in the original array. In other words, it mutates the array in place. One way to avoid this is to first concatenate an empty array to the one being sorted (remember that slice and concat return a new array), then run the sort method.

Use the sort method in the nonMutatingSort function to sort the elements of an array in ascending order. The function should return a new array, and not mutate the globalArray variable.

var globalArray = [5, 6, 3, 2, 9];

function nonMutatingSort(arr) {

// Only change code below this line

var result=[].concat(arr).sort(function(a,b){

return a-b;

});

return result;

// Only change code above this line

}

nonMutatingSort(globalArray);

Another method:

var globalArray = [5, 6, 3, 2, 9];

function nonMutatingSort(arr) {

let temp=[];

// Only change code below this line

var result=arr.concat(temp);

result.sort(function(a,b){

return a-b;

})

return result;

// Only change code above this line

}

nonMutatingSort(globalArray);

MYCode:

var globalArray = [5, 6, 3, 2, 9];

function nonMutatingSort(arr) {

var temp=[];

temp=arr.concat();

var result=temp.sort(function(a,b){

return a-b;

});

return result;

// Only change code above this line

}

console.log(nonMutatingSort(globalArray));

console.log(globalArray);

\*\*\*\*\*\*\*

**Split a String into an Array Using the split Method**

The split method splits a string into an array of strings. It takes an argument for the delimiter, which can be a character to use to break up the string or a regular expression. For example, if the delimiter is a space, you get an array of words, and if the delimiter is an empty string, you get an array of each character in the string.

Here are two examples that split one string by spaces, then another by digits using a regular expression:

var str = "Hello World";

var bySpace = str.split(" ");

var otherString = "How9are7you2today";

var byDigits = otherString.split(/\d/);

bySpace would have the value ["Hello", "World"] and byDigits would have the value ["How", "are", "you", "today"].

Since strings are immutable, the split method makes it easier to work with them.

Use the split method inside the splitify function to split str into an array of words. The function should return the array. Note that the words are not always separated by spaces, and the array should not contain punctuation.

Run the Tests

Reset All Code

Get Help

function splitify(str) {

// Only change code below this line

return str.split(/\s|\W/);

// Only change code above this line

}

console.log(splitify("Hello World,I-am code"));

//return str.split(/\s|\W/);

\*\*\*\*\*\*\*

**Combine an Array into a String Using the join Method**

The join method is used to join the elements of an array together to create a string. It takes an argument for the delimiter that is used to separate the array elements in the string.

Here's an example:

var arr = ["Hello", "World"];

var str = arr.join(" ");

str would have a value of the string Hello World.

Use the join method (among others) inside the sentensify function to make a sentence from the words in the string str. The function should return a string. For example, I-like-Star-Wars would be converted to I like Star Wars. For this challenge, do not use the replace method.

function sentensify(str) {

// Only change code below this line

return str.split(/\W+/).join(" ");

// Only change code above this line

}

console.log(sentensify("May-the-force-be-with-you"));

// running tests

May the force be with you

// tests completed

// console output

May the force be with you

\*\*\*\*\*\*\*\*\*

**Apply Functional Programming to Convert Strings to URL Slugs**

The last several challenges covered a number of useful array and string methods that follow functional programming principles. We've also learned about reduce, which is a powerful method used to reduce problems to simpler forms. From computing averages to sorting, any array operation can be achieved by applying it. Recall that map and filter are special cases of reduce.

Let's combine what we've learned to solve a practical problem.

Many content management sites (CMS) have the titles of a post added to part of the URL for simple bookmarking purposes. For example, if you write a Medium post titled Stop Using Reduce, it's likely the URL would have some form of the title string in it (.../stop-using-reduce). You may have already noticed this on the freeCodeCamp site.

Fill in the urlSlug function so it converts a string title and returns the hyphenated version for the URL. You can use any of the methods covered in this section, and don't use replace. Here are the requirements:

The input is a string with spaces and title-cased words

The output is a string with the spaces between words replaced by a hyphen (-)

The output should be all lower-cased letters

The output should not have any spaces

// Only change code below this line

function urlSlug(title) {

return title.toLowerCase().trim().split(/\W+/).join("-");

}

console.log(urlSlug("Winter Is Coming"));

// Only change code above this line

\*\*\*\*\*\*\*

**Use the every Method to Check that Every Element in an Array Meets a Criteria**

The every method works with arrays to check if *every* element passes a particular test. It returns a Boolean value - true if all values meet the criteria, false if not.

For example, the following code would check if every element in the numbers array is less than 10:

var numbers = [1, 5, 8, 0, 10, 11];

numbers.every(function(currentValue) {

return currentValue < 10;

});

The every method would return false here.

Use the every method inside the checkPositive function to check if every element in arr is positive. The function should return a Boolean value.

function checkPositive(arr) {

// Only change code below this line

return arr.every(function(currentvalue){

return currentvalue>0;

})

// Only change code above this line

}

checkPositive([1, 2, 3, -4, 5]);

\*\*\*\*\*\*\*\*

**Use the some Method to Check that Any Elements in an Array Meet a Criteria**

The some method works with arrays to check if *any* element passes a particular test. It returns a Boolean value - true if any of the values meet the criteria, false if not.

For example, the following code would check if any element in the numbers array is less than 10:

var numbers = [10, 50, 8, 220, 110, 11];

numbers.some(function(currentValue) {

return currentValue < 10;

});

The some method would return true.

Use the some method inside the checkPositive function to check if any element in arr is positive. The function should return a Boolean value.

function checkPositive(arr) {

// Only change code below this line

return arr.some(function(currentvalue){

return currentvalue>0;

})

// Only change code above this line

}

checkPositive([1, 2, 3, -4, 5]);

\*\*\*\*\*\*

**Introduction to Currying and Partial Application**

The arity of a function is the number of arguments it requires. Currying a function means to convert a function of N arity into N functions of arity 1.

In other words, it restructures a function so it takes one argument, then returns another function that takes the next argument, and so on.

Here's an example:

function unCurried(x, y) {

return x + y;

}

function curried(x) {

return function(y) {

return x + y;

}

}

const curried = x => y => x + y

curried(1)(2)

curried(1)(2) would return 3.

This is useful in your program if you can't supply all the arguments to a function at one time. You can save each function call into a variable, which will hold the returned function reference that takes the next argument when it's available. Here's an example using the curried function in the example above:

var funcForY = curried(1);

console.log(funcForY(2)); // 3

Similarly, partial application can be described as applying a few arguments to a function at a time and returning another function that is applied to more arguments. Here's an example:

function impartial(x, y, z) {

return x + y + z;

}

var partialFn = impartial.bind(this, 1, 2);

partialFn(10); // 13

Fill in the body of the add function so it uses currying to add parameters x, y, and z.

function add(x) {

// Only change code below this line

return function(y){

return function(z){

return x+y+z;

}

}

// Only change code above this line

}

/\*

const add=x=>y=>z=>x+y+z;

\*/

add(10)(20)(30);

Another method:

const add=x=>y=>z=>x+y+z;

add(10)(20)(30);

\*\*\*\*\*\*\*\*\*\*

**INTERMEDIATE ALGORITHM SCRIPTING**

**Sum All Numbers in a Range**

We'll pass you an array of two numbers. Return the sum of those two numbers plus the sum of all the numbers between them. The lowest number will not always come first.

For example, sumAll([4,1]) should return 10 because sum of all the numbers between 1 and 4 (both inclusive) is 10.

function sumAll(arr) {

let newArr = [];

if(arr[0] < arr[1]) {

for (let i = arr[0]; i <= arr[1]; i++) {

newArr.push(i);

}

} else {

for (let j = arr[0]; j >= arr[1]; j--) {

newArr.push(j);

}

}

let total = newArr.reduce(function(sum, value) {

return sum + value;

});

return total;

}

sumAll([1, 4]);

\*\*\*\*\*\*\*

**Diff Two Arrays**

Compare two arrays and return a new array with any items only found in one of the two given arrays, but not both. In other words, return the symmetric difference of the two arrays.

**Note:** You can return the array with its elements in any order.

function diffArray(arr1, arr2) {

var newArr = arr1.concat(arr2);

return newArr.filter(function(value,index,array){

if(array.slice(0,index).indexOf(value)===-1 && array.slice(index+1).indexOf(value)===-1){

return value

}

});

}

diffArray([1, 2, 3, 5], [1, 2, 3, 4, 5]);

\*\*\*\*\*\*\*

**Seek and Destroy**

You will be provided with an initial array (the first argument in the destroyer function), followed by one or more arguments. Remove all elements from the initial array that are of the same value as these arguments.

**Note:** You have to use the arguments object.

function destroyer(arr) {

let args=Array.prototype.slice.call(arguments);

args.splice(0,1);

for(let i=0;i<arr.length;i++){

for(let j=0;j<args.length;j++){

if(arr[i]===args[j]){

delete arr[i];

}

}

}

return arr.filter(Boolean);

}

destroyer([1, 2, 3, 1, 2, 3], 2, 3);

\*\*\*\*\*\*\*

**Wherefore art thou**

Make a function that looks through an array of objects (first argument) and returns an array of all objects that have matching name and value pairs (second argument). Each name and value pair of the source object has to be present in the object from the collection if it is to be included in the returned array.

For example, if the first argument is [{ first: "Romeo", last: "Montague" }, { first: "Mercutio", last: null }, { first: "Tybalt", last: "Capulet" }], and the second argument is { last: "Capulet" }, then you must return the third object from the array (the first argument), because it contains the name and its value, that was passed on as the second argument.

function whatIsInAName(collection, source) {

// What's in a name?

var arr = [];

// Only change code below this line

arr = collection.filter(function(obj) {

for (let i in source) {

if (source[i] != obj[i]) {

return false;

}

}

return true;

});

// Only change code above this line

return arr;

}

whatIsInAName([{ first: "Romeo", last: "Montague" }, { first: "Mercutio", last: null }, { first: "Tybalt", last: "Capulet" }], { last: "Capulet" });

\*\*\*\*\*\*

**Spinal Tap Case**

Convert a string to spinal case. Spinal case is all-lowercase-words-joined-by-dashes.

function spinalCase(str) {

str = str.replace(/([a-z])([A-Z])/g,"$1 $2" ).toLowerCase();

return str.replace( /\s|\_/g,"-" );

}

spinalCase('This Is Spinal Tap');

\*\*\*\*\*\*\*

**Pig Latin**

Pig Latin is a way of altering English Words. The rules are as follows:

- If a word begins with a consonant, take the first consonant or consonant cluster, move it to the end of the word, and add ay to it.

- If a word begins with a vowel, just add way at the end.

Translate the provided string to Pig Latin. Input strings are guaranteed to be English words in all lowercase.

function translatePigLatin(str) {

let result;

if(str.match(/[aeiou]+/)){

if(str.match(/(^[^aeiou]+)/)){

let sub2 = str.replace(/(^[^aeiou]+)(\w+)/,"$2$1ay");

result = sub2

}else{

result = str+"way";

}

}else{

result= str+"ay";

}

return result;

}

translatePigLatin("consonant");

\*\*\*\*\*\*\*\*

**Search and Replace**

Perform a search and replace on the sentence using the arguments provided and return the new sentence.

First argument is the sentence to perform the search and replace on.

Second argument is the word that you will be replacing (before).

Third argument is what you will be replacing the second argument with (after).

**Note:** Preserve the case of the first character in the original word when you are replacing it. For example if you mean to replace the word Book with the word dog, it should be replaced as Dog

let isUpperCase = function(letter) {

return letter.toUpperCase() === letter;

}

let myReplace = function(text, before, after) {

let correctLetter = isUpperCase(before[0]) ? after.charAt(0).toUpperCase() : after.charAt(0).toLowerCase();

let newAfter = correctLetter + after.substr(1);

return text.replace(before, newAfter);

}

console.log(myReplace("A quick brown fox jumped over the lazy dog", "jumped", "Leaped"));

console.log(myReplace("His name is Tom", "Tom", "john"));

\*\*\*\*\*\*\*\*\*\*\*

**DNA Pairing**

The DNA strand is missing the pairing element. Take each character, get its pair, and return the results as a 2d array.

[Base pairs](http://en.wikipedia.org/wiki/Base_pair) are a pair of AT and CG. Match the missing element to the provided character.

Return the provided character as the first element in each array.

For example, for the input GCG, return [["G", "C"], ["C","G"], ["G", "C"]]

The character and its pair are paired up in an array, and all the arrays are grouped into one encapsulating array.

function pairElement(str) {

const pairs = {A: 'T', T: 'A', C: 'G', G: 'C'};

return str.split('').map(function(basePairs){

return[basePairs, pairs[basePairs]];

});

}

pairElement("GCG");

\*\*\*\*\*\*\*\*

**Missing letters**

Find the missing letter in the passed letter range and return it.

If all letters are present in the range, return undefined.

function fearNotLetter(str) {

const numbers = str.split('').map(function(letter){

return letter.charCodeAt(0);

});

for (let i = 0; i < numbers.length; i++) {

if (numbers[i+1] - numbers[i] > 1) {

return String.fromCharCode(numbers[i] + 1);

}

}

}

\*\*\*\*\*\*\*

**Sorted Union**

Write a function that takes two or more arrays and returns a new array of unique values in the order of the original provided arrays.

In other words, all values present from all arrays should be included in their original order, but with no duplicates in the final array.

The unique numbers should be sorted by their original order, but the final array should not be sorted in numerical order.

Check the assertion tests for examples.

function uniteUnique(arr) {

const args=Array.prototype.slice.call(arguments);

return args.reduce(function(a,b){

return a.concat(b.filter(function(subArrayItems){

return a.indexOf(subArrayItems)<0;

}))

})

}

uniteUnique([1, 3, 2], [5, 2, 1, 4], [2, 1]);

\*\*\*\*\*\*\*\*

**Convert HTML Entities**

Convert the characters &, <, >, " (double quote), and ' (apostrophe), in a string to their corresponding HTML entities.

function convertHTML(str) {

const symbols={

"&":"&amp;",

"<":"&lt;",

">":"&gt;",

'\"':"&quot;",

"'":"&apos;"

};

return str.split('').map(function(char){

return symbols[char]||char;

}).join('');

}

convertHTML("Dolce & Gabbana");

\*\*\*\*\*\*\*\*

**Sum All Odd Fibonacci Numbers**

Given a positive integer num, return the sum of all odd Fibonacci numbers that are less than or equal to num.

The first two numbers in the Fibonacci sequence are 1 and 1. Every additional number in the sequence is the sum of the two previous numbers. The first six numbers of the Fibonacci sequence are 1, 1, 2, 3, 5 and 8.

For example, sumFibs(10) should return 10 because all odd Fibonacci numbers less than or equal to 10 are 1, 1, 3, and 5.

function sumFibs(num) {

Array.prototype.last = function() {

return this[this.length-1];

};

Array.prototype.secondToLast = function() {

return this[this.length-2];

};

const fib = [1,1];

while(fib.secondToLast() + fib.last() <= num) {

fib.push(fib.secondToLast() + fib.last());

}

return fib.filter(function(spot) {

return spot % 2 !== 0;

}).reduce(function(a,b) {

return a + b;

});

}

sumFibs(4);

\*\*\*\*\*\*\*

**Sum All Primes**

A prime number is a whole number greater than 1 with exactly two divisors: 1 and itself. For example, 2 is a prime number because it is only divisible by 1 and 2. In contrast, 4 is not prime since it is divisible by 1, 2 and 4.

Rewrite sumPrimes so it returns the sum of all prime numbers that are less than or equal to num.

function sumPrimes(num) {

const prime = [];

for(let i = 2; i <= num; i++) {

if(isPrime(i)) {

prime.push(i);

}

}

function isPrime (num) {

for (let j = 2; j < num; j++) {

if(num % j === 0) {

return false;

}

}

return true;

}

return prime.reduce(function(a,b) {

return a + b;

});

}

sumPrimes(10);

\*\*\*\*\*\*\*

**Smallest Common Multiple**

Find the smallest common multiple of the provided parameters that can be evenly divided by both, as well as by all sequential numbers in the range between these parameters.

The range will be an array of two numbers that will not necessarily be in numerical order.

For example, if given 1 and 3, find the smallest common multiple of both 1 and 3 that is also evenly divisible by all numbers *between* 1 and 3. The answer here would be 6.

function smallestCommons(arr) {

// Setup

const [min, max] = arr.sort((a, b) => a - b);

const numberDivisors = max - min + 1;

// Largest possible value for SCM

let upperBound = 1;

for (let i = min; i <= max; i++) {

upperBound \*= i;

}

// Test all multiples of 'max'

for (let multiple = max; multiple <= upperBound; multiple += max) {

// Check if every value in range divides 'multiple'

let divisorCount = 0;

for (let i = min; i <= max; i++) {

// Count divisors

if (multiple % i === 0) {

divisorCount += 1;

}

}

if (divisorCount === numberDivisors) {

return multiple;

}

}

}

smallestCommons([1, 5]);

\*\*\*\*\*\*\*\*\*

**Drop it**

Given the array arr, iterate through and remove each element starting from the first element (the 0 index) until the function func returns true when the iterated element is passed through it.

Then return the rest of the array once the condition is satisfied, otherwise, arr should be returned as an empty array.

function dropElements(arr, func) {

// Drop them elements.

while (!func(arr[0])) {

arr.shift();

}

return arr;

}

dropElements([1, 2, 3], function(n) {return n < 3; });

\*\*\*\*\*\*\*\*

**Steamroller**

Flatten a nested array. You must account for varying levels of nesting.

function steamrollArray(arr) {

let array = [];

flatten(arr);

function flatten(arr) {

arr.forEach(function(item) {

if (!Array.isArray(item)) {

array.push(item);

}

else {

flatten(item);

}

});

}

return array;

}

steamrollArray([1, [2], [3, [[4]]]]);

\*\*\*\*\*\*\*\*

**Binary Agents**

Return an English translated sentence of the passed binary string.

The binary string will be space separated.

function binaryAgent(str) {

return str.split(' ').map(function(binary) {

return String.fromCharCode(parseInt(binary,2));

}).join('');

}

console.log(binaryAgent("01000001 01110010 01100101 01101110 00100111 01110100 00100000 01100010 01101111 01101110 01100110 01101001 01110010 01100101 01110011 00100000 01100110 01110101 01101110 00100001 00111111"));

\*\*\*\*\*\*\*\*\*\*

**Everything Be True**

Check if the predicate (second argument) is truthy on all elements of a collection (first argument).

In other words, you are given an array collection of objects. The predicate pre will be an object property and you need to return true if its value is truthy. Otherwise, return false.

In JavaScript, truthy values are values that translate to true when evaluated in a Boolean context.

Remember, you can access object properties through either dot notation or [] notation.

function truthCheck(collection, pre) {

// Is everyone being true?

return collection.every(function(prop) {

return prop[pre];

});

}

truthCheck([{"user": "Tinky-Winky", "sex": "male"}, {"user": "Dipsy", "sex": "male"},

{"user": "Laa-Laa", "sex": "female"}, {"user": "Po", "sex": "female"}], "sex");

\*\*\*\*\*\*\*\*\*

**Arguments Optional**

Create a function that sums two arguments together. If only one argument is provided, then return a function that expects one argument and returns the sum.

For example, addTogether(2, 3) should return 5, and addTogether(2) should return a function.

Calling this returned function with a single argument will then return the sum:

var sumTwoAnd = addTogether(2);

sumTwoAnd(3) returns 5.

If either argument isn't a valid number, return undefined.

function addTogether() {

let check = function(num) {

return typeof num === 'number' ? num : undefined;

};

let a = check(arguments[0]);

let b = check(arguments[1]);

if (arguments.length == 2) {

return a && b ? a + b : undefined;

} else {

if (a) {

return function (nextArg) {

return a && check(nextArg) ? a + nextArg : undefined;

}

} else {

return undefined;

}

}

}

addTogether(2,3);

\*\*\*\*\*\*\*\*

**Make a Person**

Fill in the object constructor with the following methods below:

getFirstName()

getLastName()

getFullName()

setFirstName(first)

setLastName(last)

setFullName(firstAndLast)

Run the tests to see the expected output for each method. The methods that take an argument must accept only one argument and it has to be a string. These methods must be the only available means of interacting with the object.

var Person = function(firstAndLast) {

var arr = firstAndLast.split(' ');

var firstName = arr[0];

var lastName = arr[1];

// Complete the method below and implement the others similarly

this.getFullName = function() {

return firstName + ' ' + lastName;

};

this.getFirstName = function() {

return firstName;

};

this.getLastName = function() {

return lastName;

};

this.setFirstName = function(first){

firstName = first;

};

this.setLastName = function(last){

lastName = last;

};

this.setFullName = function(firstAndLast){

var arr = firstAndLast.split(' ');

firstName = arr[0];

lastName = arr[1];

};

};

var bob = new Person('Bob Ross');

bob.getFullName();

\*\*\*\*\*\*\*

**Map the Debris**

Return a new array that transforms the elements' average altitude into their orbital periods (in seconds).

The array will contain objects in the format {name: 'name', avgAlt: avgAlt}.

You can read about orbital periods [on Wikipedia](http://en.wikipedia.org/wiki/Orbital_period).

The values should be rounded to the nearest whole number. The body being orbited is Earth.

The radius of the earth is 6367.4447 kilometers, and the GM value of earth is 398600.4418 km3s-2.

function orbitalPeriod(arr) {

var GM = 398600.4418;

var earthRadius = 6367.4447;

arr.forEach(function(object) {

object.orbitalPeriod = Math.round(2\*Math.PI\*Math.sqrt(Math.pow(earthRadius + object.avgAlt,3)/GM))

delete object.avgAlt;

});

return arr;

}

orbitalPeriod([{name : "sputnik", avgAlt : 35873.5553}]);

\*\*\*\*\*\*\*\*\*\*\*

**Javascript Projects**

**Palindrome Checker**

Return true if the given string is a palindrome. Otherwise, return false.

A palindrome is a word or sentence that's spelled the same way both forward and backward, ignoring punctuation, case, and spacing.

**Note:** You'll need to remove **all non-alphanumeric characters** (punctuation, spaces and symbols) and turn everything into the same case (lower or upper case) in order to check for palindromes.

We'll pass strings with varying formats, such as racecar, RaceCar, and race CAR among others.

We'll also pass strings with special symbols, such as 2A3\*3a2, 2A3 3a2, and 2\_A3\*3#A2.

function palindrome(str) {

const left=str.replace(/[\W\_]/ig,'').toLowerCase();

const rigth=left.split('').reverse().join('');

return left===rigth;

}

console.log(palindrome("eye"));

Another method:

|  |  |
| --- | --- |
|  |  |
| SOLUTION ONE: WITHOUT EXPRESSIONS (REGEX) |  |
|  |  |
|  | function palindrome(str) { |
|  |  |
|  | str = str.toLowerCase(); |
|  | // changed string to lowercase so no sensitive case issues |
|  | for (let i = 0; i = str.length; i++) { |
|  | str = str.replace(' ', ''); |
|  | str = str.replace(',', ''); |
|  | str = str.replace('.', ''); |
|  | str = str.replace('\_', ''); |
|  | str = str.replace('/', ''); |
|  | str = str.replace('-', ''); |
|  | str = str.replace('(', ''); |
|  | str = str.replace(')', ''); |
|  | } |
|  | // loop to replace special characters without using regular expressions |
|  | const copy = str.split('').reverse().join(''); |
|  | // create a copy of our string and reverse it |
|  | if (copy === str){ |
|  | return true; |
|  | } else { |
|  | return false; |
|  | } |
|  |  |
|  | return str; |
|  | } |
|  |  |
|  | palindrome("eye"); |
|  |  |

\*\*\*\*\*\*\*\*

**Roman Numeral Converter**

Convert the given number into a roman numeral.

All [roman numerals](http://www.mathsisfun.com/roman-numerals.html) answers should be provided in upper-case.

Run the Tests

Reset All Code

Get Help

function convertToRoman(num) {

var romannumbers={

M:1000,

CM:900,

D:500,

CD:400,

C:100,

XC:90,

L:50,

XL:40,

X:10,

IX:9,

V:5,

IV:4,

I:1

},

roman='';

for(var i in romannumbers){

while(num>=romannumbers[i]){

roman +=i;

num -=romannumbers[i];

}

}

return roman;

}

convertToRoman(36);

\*\*\*\*\*\*\*\*

**Caesars Cipher**

One of the simplest and most widely known ciphers is a Caesar cipher, also known as a shift cipher. In a shift cipher the meanings of the letters are shifted by some set amount.

A common modern use is the [ROT13](https://en.wikipedia.org/wiki/ROT13) cipher, where the values of the letters are shifted by 13 places. Thus A ↔ N, B ↔ O and so on.

Write a function which takes a [ROT13](https://en.wikipedia.org/wiki/ROT13) encoded string as input and returns a decoded string.

All letters will be uppercase. Do not transform any non-alphabetic character (i.e. spaces, punctuation), but do pass them on.

function rot13(str) {

str=str.split('').map(encoding).join('');

return str;

function encoding(letter){

var code=letter.charCodeAt();

if(code>=78 && code<=90){

code -=13;

}

else if(code>=65 && code<=77){

code +=13;

}

return String.fromCharCode(code);

}

}

rot13("SERR PBQR PNZC");

\*\*\*\*\*\*\*\*

**Telephone Number Validator**

Return true if the passed string looks like a valid US phone number.

The user may fill out the form field any way they choose as long as it has the format of a valid US number. The following are examples of valid formats for US numbers (refer to the tests below for other variants):

555-555-5555

(555)555-5555

(555) 555-5555

555 555 5555

5555555555

1 555 555 5555

For this challenge you will be presented with a string such as 800-692-7753 or 8oo-six427676;laskdjf. Your job is to validate or reject the US phone number based on any combination of the formats provided above. The area code is required. If the country code is provided, you must confirm that the country code is 1. Return true if the string is a valid US phone number; otherwise return false.

function telephoneCheck(str) {

let result = /^(1\s?)?(\(\d{3}\)|\d{3})[\s\-]?\d{3}[\s\-]?\d{4}$/gm

return result.test(str);

}

telephoneCheck("555-555-5555");

\*\*\*\*\*\*\*\*\*

**Cash Register**

Design a cash register drawer function checkCashRegister() that accepts purchase price as the first argument (price), payment as the second argument (cash), and cash-in-drawer (cid) as the third argument.

cid is a 2D array listing available currency.

The checkCashRegister() function should always return an object with a status key and a change key.

Return {status: "INSUFFICIENT\_FUNDS", change: []} if cash-in-drawer is less than the change due, or if you cannot return the exact change.

Return {status: "CLOSED", change: [...]} with cash-in-drawer as the value for the key change if it is equal to the change due.

Otherwise, return {status: "OPEN", change: [...]}, with the change due in coins and bills, sorted in highest to lowest order, as the value of the change key.

|  |  |
| --- | --- |
| **Currency Unit** | **Amount** |
| Penny | $0.01 (PENNY) |
| Nickel | $0.05 (NICKEL) |
| Dime | $0.1 (DIME) |
| Quarter | $0.25 (QUARTER) |
| Dollar | $1 (ONE) |
| Five Dollars | $5 (FIVE) |
| Ten Dollars | $10 (TEN) |
| Twenty Dollars | $20 (TWENTY) |
| One-hundred Dollars | $100 (ONE HUNDRED) |

See below for an example of a cash-in-drawer array:

[

["PENNY", 1.01],

["NICKEL", 2.05],

["DIME", 3.1],

["QUARTER", 4.25],

["ONE", 90],

["FIVE", 55],

["TEN", 20],

["TWENTY", 60],

["ONE HUNDRED", 100]

]

const REGISTER\_STATUS = {closed: 'CLOSED', insufficientFunds: 'INSUFFICIENT\_FUNDS', open: 'OPEN'}

function checkCashRegister(price, cash, cid) {

let cashRegister = { status: '', change: cid};

const changeNeeded = parseFloat(cash - price).toFixed(2);

const changeAvailable = getTotalCashRegisterChange(cid);

cashRegister.status = getTotalCashRegisterStatus(changeNeeded, changeAvailable);

if (cashRegister.status === REGISTER\_STATUS.insufficientFunds) {

cashRegister.change = [];

return cashRegister;

}

cashRegister.change = getCustomerChange(changeNeeded, cid);

if (changeNeeded > getTotalCashRegisterChange(cashRegister.change)) {

cashRegister.status = REGISTER\_STATUS.insufficientFunds;

cashRegister.change = [];

}

if (cashRegister.status === REGISTER\_STATUS.closed) {

cashRegister.change = [...cid];

}

return cashRegister;

}

function getCustomerChange(changeNeeded, changeInDrawer) {

const change = [];

const currencyDictionary = {

"PENNY": 0.01,

"NICKEL": 0.05,

"DIME": 0.10,

"QUARTER": 0.25,

"ONE": 1.00,

"FIVE": 5.00,

"TEN": 10.00,

"TWENTY": 20.00,

"ONE HUNDRED": 100.00

};

for (let i = changeInDrawer.length - 1; i >= 0; i--) {

const coinName = changeInDrawer[i][0];

const coinTotal = changeInDrawer[i][1];

const coinValue = currencyDictionary[coinName];

let coinAmount = (coinTotal / coinValue).toFixed(2);

let coinsToReturn = 0;

while(changeNeeded >= coinValue && coinAmount > 0) {

changeNeeded -= coinValue;

changeNeeded = changeNeeded.toFixed(2);

coinAmount--;

coinsToReturn++;

}

if(coinsToReturn > 0) {

change.push([coinName, coinsToReturn \* coinValue]);

}

}

return change;

}

function getTotalCashRegisterStatus(changeNeeded, changeAvailable) {

if (Number(changeNeeded) > Number(changeAvailable)) {

return REGISTER\_STATUS.insufficientFunds;

}

if (Number(changeNeeded) < Number(changeAvailable)) {

return REGISTER\_STATUS.open;

}

return REGISTER\_STATUS.closed;

}

function getTotalCashRegisterChange(changeInDrawer) {

let total = 0;

for (let change of changeInDrawer) {

let changeValue = change [1];

total += changeValue;

}

return total.toFixed(2);

}

checkCashRegister(19.5, 20, [["PENNY", 1.01], ["NICKEL", 2.05], ["DIME", 3.1], ["QUARTER", 4.25], ["ONE", 90], ["FIVE", 55], ["TEN", 20], ["TWENTY", 60], ["ONE HUNDRED", 100]]);

\*\*\*\*\*\*\*\*