**Build JavaScript Objects**

You may have heard the term object before.

Objects are similar to arrays, except that instead of using indexes to access and modify their data, you access the data in objects through what are called properties.

Objects are useful for storing data in a structured way, and can represent real world objects, like a cat.

Here's a sample cat object:

const cat = {

"name": "Whiskers",

"legs": 4,

"tails": 1,

"enemies": ["Water", "Dogs"]

};

In this example, all the properties are stored as strings, such as name, legs, and tails. However, you can also use numbers as properties. You can even omit the quotes for single-word string properties, as follows:

const anotherObject = {

make: "Ford",

5: "five",

"model": "focus"

};

However, if your object has any non-string properties, JavaScript will automatically typecast them as strings.

Make an object that represents a dog called myDog which contains the properties name (a string), legs, tails and friends.

You can set these object properties to whatever values you want, as long as name is a string, legs and tails are numbers, and friends is an array.

const myDog = {

// Only change code below this line

"name": "Simon",

legs: 4,

tails: 2,

friends: ["Bicho","Wendy","Baguira","Cala",22],

// Only change code above this line

};

**Accessing Object Properties with Dot Notation**

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

Dot notation is what you use when you know the name of the property you're trying to access ahead of time.

Here is a sample of using dot notation (.) to read an object's property:

const myObj = {

prop1: "val1",

prop2: "val2"

};

const prop1val = myObj.prop1;

const prop2val = myObj.prop2;

prop1val would have a value of the string val1, and prop2val would have a value of the string val2.

Read in the property values of testObj using dot notation. Set the variable hatValue equal to the object's property hat and set the variable shirtValue equal to the object's property shirt.

// Setup

const testObj = {

"hat": "ballcap",

"shirt": "jersey",

"shoes": "cleats"

};

// Only change code below this line

const hatValue = testObj; // Change this line

const shirtValue = testObj; // Change this line

—-------

// Setup

const testObj = {

"hat": "ballcap",

"shirt": "jersey",

"shoes": "cleats"

};

// Only change code below this line

const hatValue = testObj.hat; // Change this line

const shirtValue = testObj.shirt; // Change this line

**Accessing Object Properties with Bracket Notation**

The second way to access the properties of an object is bracket notation ([]). If the property of the object you are trying to access has a space in its name, you will need to use bracket notation.

However, you can still use bracket notation on object properties without spaces.

Here is a sample of using bracket notation to read an object's property:

const myObj = {

"Space Name": "Kirk",

"More Space": "Spock",

"NoSpace": "USS Enterprise"

};

myObj["Space Name"];

myObj['More Space'];

myObj["NoSpace"];

myObj["Space Name"] would be the string Kirk, myObj['More Space'] would be the string Spock, and myObj["NoSpace"] would be the string USS Enterprise.

Note that property names with spaces in them must be in quotes (single or double).

Read the values of the properties an entree and the drink of testObj using bracket notation and assign them to entreeValue and drinkValue respectively.

// Setup

const testObj = {

"an entree": "hamburger",

"my side": "veggies",

"the drink": "water"

};

// Only change code below this line

const entreeValue = testObj; // Change this line

const drinkValue = testObj; // Change this line

// Setup

const testObj = {

"an entree": "hamburger",

"my side": "veggies",

"the drink": "water",

"payed": 33,

pay : 5

};

// Only change code below this line

const entreeValue = testObj["an entree"]; // Change this line

const drinkValue = testObj["the drink"]; // Change this line

//Mis pruebas

console.log(testObj["payed"]);

console.log(testObj["pay"]);

//console.log(testObj[pay]); //Da error

console.log(testObj.pay);

console.log(testObj.payed);

**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:

const dogs = {

Fido: "Mutt",

Hunter: "Doberman",

Snoopie: "Beagle"

};

const myDog = "Hunter";

const 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:

const someObj = {

propName: "John"

};

function propPrefix(str) {

const s = "prop";

return s + str;

}

const 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

const testObj = {

12: "Namath",

16: "Montana",

19: "Unitas"

};

// Only change code below this line

const playerNumber = 42; // Change this line

const player = testObj; // Change this line

—--------

// Setup

const testObj = {

12: "Namath",

16: "Montana",

19: "Unitas"

};

// Only change code below this line

const playerNumber = 16; // Change this line

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

console.log(player);

**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:

const 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

const myDog = {

"name": "Coder",

"legs": 4,

"tails": 1,

"friends": ["freeCodeCamp Campers"]

};

// Only change code below this line

\_\_\_\_\_\_\_\_\_

// Setup

const myDog = {

"name": "Coder",

"legs": 4,

"tails": 1,

"friends": ["freeCodeCamp Campers"]

};

// Only change code below this line

console.log(myDog.name);

myDog.name = "Happy Coder";

console.log(myDog.name);

**Add New Properties to a JavaScript Object**

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:

const 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

const myDog = {

"name": "Happy Coder",

"legs": 4,

"tails": 1,

"friends": ["freeCodeCamp Campers"]

};

\_\_\_\_

const myDog = {

"name": "Happy Coder",

"legs": 4,

"tails": 1,

"friends": ["freeCodeCamp Campers"]

};

myDog.bark = "woof";

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

**Delete Properties from a JavaScript Object**

We can also delete properties from objects like this:

delete ourDog.bark;

Example:

const 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

const myDog = {

"name": "Happy Coder",

"legs": 4,

"tails": 1,

"friends": ["freeCodeCamp Campers"],

"bark": "woof"

};

// Only change code below this line

\_\_\_\_

// Setup

const myDog = {

"name": "Happy Coder",

"legs": 4,

"tails": 1,

"friends": ["freeCodeCamp Campers"],

"bark": "woof"

};

// Only change code below this line

delete myDog["tails"];

myDog.friends.push("my");

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

myDog.friends.unshift("Happy");

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

console.log(myDog.friends[0]);

console.log(myDog);

\_\_

CONSOLE

[ 'freeCodeCamp Campers', 'my' ]

[ 'Happy', 'freeCodeCamp Campers', 'my' ]

Happy

{ name: 'Happy Coder',

legs: 4,

friends: [ 'Happy', 'freeCodeCamp Campers', 'my' ],

bark: 'woof' }

**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:

const alpha = {

1:"Z",

2:"Y",

3:"X",

4:"W",

...

24:"C",

25:"B",

26:"A"

};

alpha[2];

alpha[24];

const 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) {

let result = "";

// Only change code below this line

switch(val) {

case "alpha":

result = "Adams";

break;

case "bravo":

result = "Boston";

break;

case "charlie":

result = "Chicago";

break;

case "delta":

result = "Denver";

break;

case "echo":

result = "Easy";

break;

case "foxtrot":

result = "Frank";

}

// Only change code above this line

return result;

}

phoneticLookup("charlie");

\_\_\_\_

// Setup

function phoneticLookup(val) {

let result = "";

// Only change code below this line

const lookup = {

"alpha": "Adams",

"bravo":"Boston",

"charlie":"Chicago",

"delta":"Denver",

"echo":"Easy",

"foxtrot":"Frank"

}

// Only change code above this line

result = lookup[val];

console.log(result);

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**

const 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".

function checkObj(obj, checkProp) {

// Only change code below this line

return "Change Me!";

// Only change code above this line

}

\_\_\_\_

function checkObj(obj, checkProp) {

// Only change code below this line

if(obj.hasOwnProperty(checkProp)){

return obj[checkProp];

}

else{

return "Not Found";

}

// Only change code above this line

}

let obj2 = {

"fuerte": "viva",

"undir": "sigueviva"

}

console.log(checkObj(obj2,"fuerte"));

console.log(checkObj(obj2,"fu"));

\_

Console:

viva

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:

const 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.

const myMusic = [

{

"artist": "Billy Joel",

"title": "Piano Man",

"release\_year": 1973,

"formats": [

"CD",

"8T",

"LP"

],

"gold": true

}

];

console.log(myMusic);

myMusic.push(

{

"artist": "Robe",

"title": "Mayeutica",

"release\_year": 2021,

"formats": [

"CD",

"8T",

"LP"

]

}

)

console.log("myMusicII");

console.log(myMusic);

—----------------------

* **SOBRE JSON**

JavaScript Object Notation

<https://www.json.org/json-en.html>

**Accessing Nested Objects**

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

Here is a nested object:

const 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.

const myStorage = {

"car": {

"inside": {

"glove box": "maps",

"passenger seat": "crumbs"

},

"outside": {

"trunk": "jack"

}

}

};

const gloveBoxContents = undefined;

const myStorage = {

"car": {

"inside": {

"glove box": "maps",

"passenger seat": "crumbs"

},

"outside": {

"trunk": "jack"

}

}

};

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

console.log(gloveBoxContents);

**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:

const 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.

const myPlants = [

{

type: "flowers",

list: [

"rose",

"tulip",

"dandelion"

]

},

{

type: "trees",

list: [

"fir",

"pine",

"birch"

]

}

];

const secondTree = "";

—-------

const myPlants = [

{

type: "flowers",

list: [

"rose",

"tulip",

"dandelion"

]

},

{

type: "trees",

list: [

"fir",

"pine",

"birch"

]

}

];

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

console.log(secondTree);

—-

**Record Collection**

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.

Run the Tests (Ctrl + Enter)

Reset All Code

Get Help

// Setup

const 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(records, id, prop, value) {

return records;

}

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

\_\_\_\_\_\_\_\_\_

// Configuración

const 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'

}

};

// Cambia solo el código debajo de esta línea

function updateRecords(records, id, prop, value) {

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

records[id][prop] = value;

}else if (prop === "tracks" && !records[id].hasOwnProperty("tracks")){

records[id][prop] = [value];

}else if(prop === "tracks"&& value !== ""){

records[id][prop].push(value);

}else if(value === ""){

delete records[id][prop];

}

return records;

}

updateRecords(recordCollection, 5439, '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.

const ourArray = [];

let 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

const myArray = [];

// Only change code below this line

// Setup

const myArray = [];

// Only change code below this line

////////

let i = 5;

while (i >= 0) {

myArray.push(i);

i--;

console.log(i);

console.log(myArray);

}

-**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 initialization 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.

const ourArray = [];

for (let 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

const myArray = [];

// Only change code below this line

// Setup

const myArray = [];

// Only change code below this line

for(let i = 1; i<=5; i++){

myArray.push(i);

console.log(myArray);

}

**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.

const ourArray = [];

for (let 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.

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

ourArray.push(i);

}

// Setup

const myArray = [];

// Only change code below this line

—---

// Setup

const myArray = [];

// Only change code below this line

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

myArray.push(i);

console.log(myArray);

}

**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.

const ourArray = [];

for (let 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

const myArray = [];

// Only change code below this line

—----

// Setup

const myArray = [];

// Only change code below this line

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

myArray.push(i);

console.log(myArray);

}

**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:

const arr = [10, 9, 8, 7, 6];

for (let 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

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

// Only change code below this line

// Setup

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

// Only change code below this line

let total = 0;

for (let i = 0; i <= myArr.length -1; i++) {

console.log(myArr[i]);

total = total + myArr[i];

console.log("Este es el resultado del array + total = " + total);

}

console.log("Este es el resultado del array + total = " + total);

/\*const arr = [10, 9, 8, 7, 6];

console.log(arr.length);

console.log(arr[5]);

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

console.log(arr[i]);

}\*/

**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:

const arr = [

[1, 2], [3, 4], [5, 6]

];

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

for (let 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) {

let product = 1;

// Only change code below this line

// Only change code above this line

return product;

}

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

—-

function multiplyAll(arr) {

var product = 1;

// Only change code below this line

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

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

//console.log(arr[i][j]);

//console.log(product);

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

//console.log("EL producto de los numeros de los subarrais = " + product);

}

}

// Only change code above this line

return product;

}

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

console.log("EL producto de la funcion = " + multiplyAll([[1, 2], [3, 4], [5, 6, 7]]));

**Iterate with JavaScript Do...While Loops**

The next type of loop you will learn is called a do...while loop. It is called a do...while loop because it will first do one pass of the code inside the loop no matter what, and then continue to run the loop while the specified condition evaluates to true.

const ourArray = [];

let i = 0;

do {

ourArray.push(i);

i++;

} while (i < 5);

The example above behaves similar to other types of loops, and the resulting array will look like [0, 1, 2, 3, 4]. However, what makes the do...while different from other loops is how it behaves when the condition fails on the first check. Let's see this in action: Here is a regular while loop that will run the code in the loop as long as i < 5:

const ourArray = [];

let i = 5;

while (i < 5) {

ourArray.push(i);

i++;

}

In this example, we initialize the value of ourArray to an empty array and the value of i to 5. When we execute the while loop, the condition evaluates to false because i is not less than 5, so we do not execute the code inside the loop. The result is that ourArray will end up with no values added to it, and it will still look like [] when all of the code in the example above has completed running. Now, take a look at a do...while loop:

const ourArray = [];

let i = 5;

do {

ourArray.push(i);

i++;

} while (i < 5);

In this case, we initialize the value of i to 5, just like we did with the while loop. When we get to the next line, there is no condition to evaluate, so we go to the code inside the curly braces and execute it. We will add a single element to the array and then increment i before we get to the condition check. When we finally evaluate the condition i < 5 on the last line, we see that i is now 6, which fails the conditional check, so we exit the loop and are done. At the end of the above example, the value of ourArray is [5]. Essentially, a do...while loop ensures that the code inside the loop will run at least once. Let's try getting a do...while loop to work by pushing values to an array.

Change the while loop in the code to a do...while loop so the loop will push only the number 10 to myArray, and i will be equal to 11 when your code has finished running.

// Setup

const myArray = [];

let i = 10;

// Only change code below this line

while (i < 5) {

myArray.push(i);

i++;

}

—

// Setup

const myArray = [];

let i = 10;

// Only change code below this line

do{

myArray.push(i);

i++

} while(i < 10);

console.log(myArray);

console.log(i);

**Replace Loops using Recursion**

Recursion is the concept that a function can be expressed in terms of itself. To help understand this, start by thinking about the following task: multiply the first n elements of an array to create the product of those elements. Using a for loop, you could do this:

function multiply(arr, n) {

let product = 1;

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

product \*= arr[i];

}

return product;

}

However, notice that multiply(arr, n) == multiply(arr, n - 1) \* arr[n - 1]. That means you can rewrite multiply in terms of itself and never need to use a loop.

function multiply(arr, n) {

if (n <= 0) {

return 1;

} else {

return multiply(arr, n - 1) \* arr[n - 1];

}

}

The recursive version of multiply breaks down like this. In the base case, where n <= 0, it returns 1. For larger values of n, it calls itself, but with n - 1. That function call is evaluated in the same way, calling multiply again until n <= 0. At this point, all the functions can return and the original multiply returns the answer.

**Note:** Recursive functions must have a base case when they return without calling the function again (in this example, when n <= 0), otherwise they can never finish executing.

Write a recursive function, sum(arr, n), that returns the sum of the first n elements of an array arr.

function sum(arr, n) {

// Only change code below this line

// Only change code above this line

}

–

////Trabajo propio previo a la leccion

function sumParadigma(myArr, x) {

let myArr2 = [];

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

myArr2.push(myArr[i]\*x);

}

console.log(myArr);

console.log(myArr2);

return myArr2;

}

let myArra = [3,6,8];

sumParadigma(myArra, 2);

console.log(sumParadigma(myArra, 3));

///Trabajo para entender esto

let myArra = [3,6,8,9];

multiply(myArra, 3);

function multiply2(arr, n) {

if (n <= 0) {

return 1;

} else {

console.log("cuanto vale aqui n = "+ n);

return multiply2(arr, n - 1) \* arr[n - 1];

}

//2 ej primero hace un paso por ella misma en "arr,n-1" multiplicando por el valor de [1]=6,luego vuelve a subir siendo 1 multiplicando por valor de [0]=3

}

console.log(multiply2(myArra, 4));

console.log(multiply2(myArra, 0));

console.log(multiply2(myArra, 1));

console.log(multiply2(myArra, 2));

—

RESULTADO DEL EJERCICIO:

function sum(arr, n) {

// Only change code below this line

if (n <= 0){

return 0;

} else {

return sum(arr, n - 1) + arr[n - 1];

}

// Only change code above this line

}

let myArra = [2,33,45,6];

console.log(sum(myArra, 2));

**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.

—---

—--

//Mi primer código para resolver el problema aún no conseguido

// Setup

const 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( let i = 0; i< contacts.length; i++){

if(contacts[i]["firstName"] === name && contacts[i].hasOwnProperty(prop)){

return contacts[i][prop];}

// }else if(contacts[i]["firstName"] !== name){

// let answerI = "No such contact";

// return answerI;

// }else if(!contacts[i].hasOwnProperty(prop)){

// let answerII = "No such property";

// return answerII;

// }

}

}

console.log(lookUpProfile("Kristian", "lastName"));

// Only change code above this line

lookUpProfile("Akira", "likes");

—Resultado

// Setup

const 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

let answerI = "No such property"

let answerII = "No such contact";

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

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

if(contacts[i].hasOwnProperty(prop)){

return contacts[i][prop];

} else {

return answerI; }

}

// else{return answerII}

}

return answerII;

// Only change code above this line

}

console.log(lookUpProfile("Kristian", "lastName"));

console.log(lookUpProfile("Akira", "address"));

lookUpProfile("Akira", "likes");

**Generate Random Fractions with JavaScript**

Random numbers are useful for creating random behavior.

JavaScript has a Math.random() function that generates a random decimal number between 0 (inclusive) and 1 (exclusive). Thus Math.random() can return a 0 but never return a 1.

**Note:** Like [Storing Values with the Assignment Operator](https://www.freecodecamp.org/learn/javascript-algorithms-and-data-structures/basic-javascript/storing-values-with-the-assignment-operator), all function calls will be resolved before the return executes, so we can return the value of the Math.random() function.

Change randomFraction to return a random number instead of returning 0.

function randomFraction() {

// Only change code below this line

return 0;

// Only change code above this line

}

—

function randomFraction() {

// Only change code below this line

let a = Math.random();

return a;

// Only change code above this line

}

console.log(randomFraction());

console.log(randomFraction());

console.log(randomFraction());

**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.random();

}

—-

function randomWholeNum() {

// Only change code below this line

let b = Math.floor(Math.random()\* 10);

return b ;

}

console.log(randomWholeNum());

**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 0;

// Only change code above this line

}

function randomRange(myMin, myMax) {

// Only change code below this line

let a = Math.floor(Math.random() \* (myMax - myMin + 1)) + myMin;

return a;

// Only change code above this line

}

console.log(randomRange(30,50));

**Use the parseInt Function**

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

const 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) {

}

convertToInteger("56");

—

function convertToInteger(str) {

let i = parseInt(str)

return i;

}

convertToInteger("56");

console.log(convertToInteger("56") + 33)

**Use the parseInt Function with a Radix**

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:

const 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) {

}

convertToInteger("10011");

—

function convertToInteger(str) {

let b = parseInt(str,2)

return b;

}

convertToInteger("10011");

console.log(convertToInteger("10011"));

**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 or equal";

}

}

This can be re-written using the conditional operator:

function findGreater(a, b) {

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

}

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) {

}

checkEqual(1, 2);

–

function checkEqual(a, b) {

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

}

checkEqual(1, 2);

console.log(checkEqual(1, 2));

console.log(checkEqual(1, 1));

**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) {

}

checkSign(10);

–

function checkSign(num) {

return num === 0 ? "zero"

: num > 0 ? "positive"

: "negative";

}

checkSign(10);

console.log(checkSign(10));

console.log(checkSign(-10));

console.log(checkSign(0));

**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.

Mis Pruebas:

// Only change code below this line

function countdown(n){

let arr = [];

for ( n; 0 < n ; n--){

arr.push(n);

}

return arr;

}

// Only change code above this line

console.log(countdown(4));

El ejercicio sin resolver

// Only change code below this line

function countdown(n){

return;

}

// Only change code above this line

—- Resuelto

// Only change code below this line

function countdown(n){

if (n < 1) {

return [];

}else {

const myArr = countdown(n - 1);

myArr.unshift(n);

return myArr;

}

}

console.log(countdown(6));

// Only change code above this line

// function countup(n) {

// if (n < 1) {

// return [];

// } else {

// const countArray = countup(n - 1);

// countArray.unshift(n);

// return countArray;

// }

// }

// console.log(countup(5));

**Use Recursion to Create a Range of Numbers**

Continuing from the previous challenge, we provide you another opportunity to create a recursive function to solve a problem.

We have defined a function named rangeOfNumbers with two parameters. The function should return an array of integers which begins with a number represented by the startNum parameter and ends with a number represented by the endNum parameter. The starting number will always be less than or equal to the ending number. Your function must use recursion by calling itself and not use loops of any kind. It should also work for cases where both startNum and endNum are the same.

function rangeOfNumbers(startNum, endNum) {

return [];

};

RESULTADO RECURSIVO

function rangeOfNumbers(startNum, endNum) {

let a = endNum - startNum;

if( a < 0){

return [];

}else {

let rangeArr = rangeOfNumbers(startNum, endNum -1)

rangeArr.push(a + startNum)

return rangeArr;

}

};

console.log(rangeOfNumbers(6, 13));

console.log(rangeOfNumbers(6, 6));

RESULTADO NORMAL

function rangeOfNumbers(startNum, endNum) {

let b = endNum - startNum;

let rangeArr = [];

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

rangeArr.push(i + startNum);

}

return rangeArr;

};

console.log(rangeOfNumbers(5, 12));

console.log(rangeOfNumbers(6, 9));

console.log(rangeOfNumbers(4, 4));