**Scope**

Scope defines where variables can be accessed or referenced. While some variables can be accessed from anywhere within a program, other variables may only be available in a specific context.

# Blocks and Scope

A block is the code found inside a set of curly braces {}. Blocks help us group one or more statements together and serve as an important structural marker for our code.

const logSkyColor = () => {

let color = 'blue';

console.log(color); // blue

};

Observe the block in an if statement:

if (dusk) {

let color = 'pink';

console.log(color); // pink

};

# Global Scope

In global scope, variables are declared outside of blocks. These variables are called global variables. Because global variables are not bound inside a block, they can be accessed by any code in the program, including code in blocks.

//variable outside the block

const city='New York City';

const logCitySkyline=() =>{

//variable inside the block

let skyscraper= 'Empire State Building';

return 'The stars over the ' + skyscraper + ' in ' + city;

}

console.log(logCitySkyline());

//output: The stars over the Empire State Building in New York City

Example:

let satellite='The Moon';

let galaxy='The Milky Way';

let stars ='North Star';

const callMyNightSky=function(){

return 'Night Sky: ' + satellite + ', ' + stars + ', and ' + galaxy;

}

console.log(callMyNightSky());

# Block Scope

When a variable is defined inside a block, it is only accessible to the code within the curly braces {}. We say that variable has block scope because it is only accessible to the lines of code within that block are called local variables.

function logVisibleLightWaves(){

const lightWaves ='Moonlight';

console.log(lightWaves);

}

logVisibleLightWaves();

console.log(lightWaves); //ReferenceError: lightWaves is not defined at Object.<anonymous>

# Scope Pollution

When you declare global variables, they go to the global namespace. The global namespace allows the variables to be accessible from anywhere in the program. These variables remain there until the program finishes which means our global namespace can fill up really quickly.

Scope pollution is when we have too many global variables that exist in the global namespace, or when we reuse variables across different scopes. Scope pollution makes it difficult to keep track of our different variables and sets us up for potential accidents. For example, globally scoped variables can collide with other variables that are more locally scoped, causing unexpected behavior in our code.

let num = 50;

const logNum = () => {

num = 100; // Take note of this line of code

console.log(num);

};

logNum(); // Prints 100

console.log(num); // Prints 100

* we want to declare a new variable but forgot to use the let keyword.

const satellite = 'The Moon';

const galaxy = 'The Milky Way';

let stars = 'North Star';

const callMyNightSky = () => {

//overwrites value

stars = 'Sirius';

    return 'Night Sky: ' + satellite + ', ' + stars + ', ' + galaxy;

};

console.log(callMyNightSky());

console.log(stars);

**Practice Good Scoping**

Tightly scoping your variables will greatly improve your code in several ways:

* It will make your code more legible since the blocks will organize your code into discrete sections.
* code more understandable since it clarifies which variables are associated with different parts of the program
* easier to maintain your code
* It will save memory in your code because it will cease to exist after the block finishes running.

const logSkyColor = () => {

const dusk = true;

let color = 'blue';

if (dusk) {

let color = 'pink';

console.log(color); // pink

}

console.log(color); // blue

};

console.log(color); // ReferenceError

Block scope is a powerful tool .since it allows us to define variables with precision(Accuracy), and not pollute the global namespace. If a variable does not need to exist outside a block— it shouldn’t!

const logVisibleLightWaves = () => {

let lightWaves = 'Moonlight';

  let region = 'The Arctic';

// Add if statement here:

if (region === 'The Arctic'){

let lightWaves = 'Northern Lights';

console.log(lightWaves);

}

console.log(lightWaves);

};

logVisibleLightWaves();

**Arrays**

One way we organize data in real life is by making lists. Let's make one here:

New Year's Resolutions:

1. Keep a journal

2. Take a falconry class

3. Learn to juggle

Lets , make this list in Javascript using Array

let newYearsResolutions = ['Keep a journal', 'Take a falconry class', 'Learn to juggle'];

console.log(newYearsResolutions);

Arrays can store any data types (including strings, numbers, and booleans). Like lists, arrays are ordered.

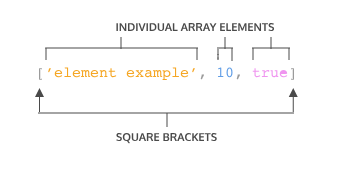
Array of the concepts:-

let concepts = ['creating arrays', 'array structures', 'array manipulation']

**Create an Array**

We, can create an array is to use an *array literal*. An array literal creates an array by wrapping items in square brackets [].

we can have an array that holds all the same data types or an array that holds different data types.



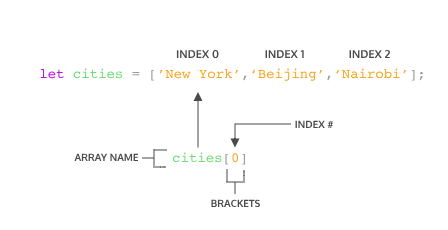
let Resolutions = ['Keep a journal', 'Take a falconry class', 'Learn to juggle'];

console.log(Resolutions);

**Accessing Elements**

Each element in an array has a numbered position known as its *index*. We can access individual items using their index.

Arrays in JavaScript are *zero-indexed*, meaning the positions start counting from 0 rather than 1.



You can also access individual characters in a string using bracket notation and the index.

const hello = 'Hello World';

console.log(hello[6]);

// Output: W

Example:

const famousSayings = ['Fortune favors the brave.', 'A joke is a very serious thing.', 'Where there is love there is life.'];

const listItem= famousSayings[0];

console.log(listItem)

console.log(famousSayings[2])

//what happens if you try to access an index that is beyond the last element?

console.log(famousSayings[3])

/\*

output:

Fortune favors the brave.

Where there is love there is life.

undefined

\*/

**Update Elements**

Once you have access to an element in an array, you can update its value.

let seasons = ['Winter', 'Spring', 'Summer', 'Fall'];

seasons[3] = 'Autumn';

console.log(seasons);

//Output: ['Winter', 'Spring', 'Summer', 'Autumn']

**Arrays with let and const**

You may recall that you can declare variables with both the let and const keywords. Variables declared with let can be reassigned and declared with the const keyword cannot be reassigned.

However, elements in an array declared with const remain mutable. Meaning that we can change the contents of a const array, but cannot reassign a new array or a different value.

let condiments = ['Ketchup', 'Mustard', 'Soy Sauce', 'Sriracha'];

console.log(condiments);

// [ 'Ketchup', 'Mustard', 'Soy Sauce', 'Sriracha' ]

condiments [0] = 'Mayo';

console.log(condiments);

//[ 'Mayo', 'Mustard', 'Soy Sauce', 'Sriracha' ]

condiments=['Mayo'];

console.log(condiments);

//[ 'Mayo' ]

const utensils = ['Fork', 'Knife', 'Chopsticks', 'Spork'];

console.log(utensils);

utensils [3] = 'Spoon';

console.log(utensils);

//utensils = ['Spoon'];

//console.log(utensils);

//error

**The .length property**

Array's built-in properties is lengthand it returns the number of items in the array.

const newYearsResolutions = ['Keep a journal', 'Take a falconry class'];

console.log(newYearsResolutions.length);

// Output: 2

**The .push() Method**

These methods are specifically called on arrays to make common tasks, like adding and removing elements, more straightforward.

One method, .push() allows us to add items to the end of an array. Here is an example of how this is used:

const itemTracker = ['item 0', 'item 1', 'item 2'];

itemTracker.push('item 3', 'item 4');

console.log(itemTracker);

// Output: ['item 0', 'item 1', 'item 2', 'item 3', 'item 4'];

**The .pop() Method**

Another array method, .pop(), removes the last item of an array.

const newItemTracker = ['item 0', 'item 1', 'item 2'];

const removed = newItemTracker.pop();

console.log(newItemTracker);

// Output: [ 'item 0', 'item 1' ]

console.log(removed);

// Output: item 2

**More Array Methods**

There are many more array methods than just .push() and .pop(). You can read about all of the array methods that exist on the

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/shift>

Some arrays methods that are available to JavaScript developers include: .join(), .slice(), .splice(), .shift(), .unshift(), and .concat()amongst many others. Using these built-in methods make it easier to do some common tasks when working with arrays.

const groceryList = ['orange juice', 'bananas', 'coffee beans', 'brown rice', 'pasta', 'coconut oil', 'plantains'];

//console.log(groceryList);

//remove 0 index value element

groceryList.shift();

//console.log(groceryList);

// add new element in the 0 index value

groceryList.unshift('popcorn');

//console.log(groceryList);

// copied some element to another array orignal array not modified

console.log(groceryList.slice(1, 4));

console.log(groceryList);

// used to find a item using index of an array

const pastaIndex = groceryList.indexOf('pasta');

console.log(pastaIndex);

//The concat() method is used to merge two or more arrays. This method does not change the existing arrays, but instead returns a new array.

var array1 = ['a', 'b', 'c'];

var array2 = ['d', 'e', 'f'];

console.log(array1.concat(array2));

//The join() method creates and returns a new string by concatenating all of the elements in an array (or an array-like object), separated by commas or a specified separator string. If the array has only one item, then that item will be returned without using the separator.

console.log(array1.join('-'));

//The splice() method changes the contents of an array by removing or replacing existing elements and/or adding new elements.

var months = ['Jan', 'March', 'April', 'June'];

months.splice(1, 0, 'Feb');

// inserts at 1st index position

console.log(months);

// expected output: Array ['Jan', 'Feb', 'March', 'April', 'June']

months.splice(4, 1, 'May');

// replaces 1 element at 4th index

console.log(months);

// expected output: Array ['Jan', 'Feb', 'March', 'April', 'May']

**Arrays and Functions**

arrays being mutable, or changeable. Well what happens if we try to change an array inside a function? Does the array keep the change after the function call or is it scoped to inside the function?

const flowers = ['peony', 'daffodil', 'marigold'];

function addFlower(arr) {

arr.push('lily');

}

addFlower(flowers);

console.log(flowers); // Output: ['peony', 'daffodil', 'marigold', 'lily']

So when you pass an array into a function, if the array is mutated inside the function, that change will be maintained outside the function as well. You might also see this concept explained as *pass-by-reference* since what we're actually passing the function is a reference to where the variable memory is stored and changing the memory.

const concept = ['arrays', 'can', 'be', 'mutated'];

function changeArr(arr){

arr[3] = 'MUTATED';

}

changeArr(concept);

console.log(concept);

function removeElement(newArr){

newArr.pop();

}

removeElement(concept);

console.log(concept);

**Nested Arrays**

When an array contains another array it is known as a *nested array*.

To access the nested arrays we can use bracket notation with the index value, just like we did to access any other element:

const nestedArr = [[1], [2, 3]];

console.log(nestedArr[1]); // Output: [2, 3]

if we wanted to access the elements within the nested array we can chain, or add on, more bracket notation with index values.

const nestedArr = [[1], [2, 3]];

console.log(nestedArr[1]); // Output: [2, 3]

console.log(nestedArr[1][0]); // Output: 2

Example:

let numberClusters=[[1,2],[3,4],[5,6]];

const target=numberClusters[2][1];

console.log(target);

**Loops**

A *loop* is a programming tool that repeats a set of instructions until a specified condition, called a *stopping condition* is reached. iterate term refers to loops simply means "to repeat".

When we need to reuse a task in our code, we often bundle that action in a function. Similarly, when we see that a process has to repeat multiple times in a row, we write a loop.

**Repeating Tasks Manually**

The best way to do that is by showing you how cumbersome(complicated) it would be if a repeated task required you to type out the same code every single time.

let vacationSpots=['one','two','three'];

console.log(vacationSpots[0]);

console.log(vacationSpots[1]);

console.log(vacationSpots[2]);

//Now imagine that the vacation list had 100 places on it— logging each array element to the console by hand would be a tedious task!

**The For Loop**

Instead of writing out the same code over and over, loops allow us to tell computers to repeat a given block of code on its own.

The typical for loop includes an *iterator variable* that usually appears in all three expressions.

A for loop contains three expressions separated by ; inside the parentheses:

1. an *initialization* starts the loop and can also be used to declare the iterator variable.
2. a *stopping condition* is the condition that the iterator variable is evaluated against— if the condition evaluates to true the code block will run, and if it evaluates to false the code will stop.
3. an *iteration statement* is used to update the iterator variable on each loop.

for(let count=5; count<=10; count++ )

{

console.log(count);

}

Output:

->This for loop makes it possible to write 5, 6, 7, and 8,9,10 programmatically.

**Looping in Reverse**

What if we want the for loop to log 3, 2, 1, and then 0?

* Set the iterator variable to the highest desired value in the initialization expression.
* Set the stopping condition for when the iterator variable is less than the desired amount.
* The iterator should decrease in intervals after each iteration.

for (let counter = 3; counter >=0; counter--){

console.log(counter);

}

**Looping through Arrays**

To loop through each element in an array, a forloop should use the array's .length property in its condition.

const vacationSpots = ['Bali', 'Paris', 'Tulum'];

// Write your code below

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

console.log('I would love to visit ' + vacationSpots[i]);

}

Output:

i would like to visit Bali

i would like to visit Paris

i would like to visit Tulum

Remember that arrays are zero-indexed, the index of the last element of an array is equivalent to the length of that array minus 1. If we tried to access an element at the index of animals.length we will have gone too far!

**Nested Loops**

When we have a loop running inside another loop, we call that a *nested loop*. One use for a nested for loop is to compare the elements in two arrays.

let bobsFollowers = ['sonu', 'kaku', 'saubhya', 'dhanu'];

let tinasFollowers = ['sonu', 'saubhya', 'Elle'];

let mutualFollowers = [];

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

{

for (let j = 0; j < tinasFollowers.length; j++)

{

if (bobsFollowers[i] === tinasFollowers[j])

{

mutualFollowers.push(bobsFollowers[i]);

}

}

};

console.log(mutualFollowers);

Example:2

const animal = 'cat';

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

console.log(animal[i]);

for (let j = 1; j < 4; j++) {

console.log(j);

}

}

Output: c 1 2 3 a 1 2 3 t 1 2 3

**The While Loop**

The while loop. To start, let's convert a for loop into a while loop:

// A for loop that prints 1, 2, and 3

for (let counterOne = 1; counterOne < 4; counterOne++){

console.log(counterOne);

}

// A while loop that prints 1, 2, and 3

let counterTwo = 1;

while (counterTwo < 4) {

console.log(counterTwo);

counterTwo++;

}

What would happen if we didn't increment counterTwo inside our block? If we didn't include this, counterTwo would always have its initial value, 0. That would mean the testing condition counterTwo < 4 would always evaluate to trueand our loop would never stop running! This is called an *infinite loop* and it's something we always want to **avoid**.

Example: Think of eating like a while loop: when you start taking bites, you don't know the exact number you'll need to become full. Rather you'll eat whileyou're hungry. In situations when we want a loop to execute an undetermined number of times, while loops are the best choice.

const cards = ['diamond', 'spade', 'heart', 'club'];

// Write your code below

let currentCard;

while (currentCard!='spade')

{

currentCard = cards[Math.floor(Math.random() \* 4)];

//Math.floor(Math.random() \* 4) will give us a random number from 0 to 3. We'll use this number to index the cards array, and assign the value of currentCard to a random element from that array.

console.log(currentCard)

}

**Do...While Statements**

In some cases, you want a piece of code to run at least once and then loop based on a specific condition after its initial run. A do...while statement says to do a task once and then keep doing it until a specified condition is no longer met.

let cupsOfSugarNeeded = 3;

let cupsAdded = 0;

do {

cupsAdded++

console.log(cupsAdded)

} while (cupsAdded < cupsOfSugarNeeded);

First, the code block after the do keyword is executed once. Then the condition is evaluated. If the condition evaluates to true, the block will execute again. The looping stops when the condition evaluates to false.

const firstMessage = 'I will print!';

const secondMessage = 'I will not print!';

// A do while with a stopping condition that evaluates to false

do {

console.log(firstMessage)

} while (true === false);

// A while loop with a stopping condition that evaluates to false

while (true === false){

console.log(secondMessage)

};

**The break Keyword**

Imagine we're looking to adopt a dog. We plan to go to the shelter every day for a year But what if we meet our dream dog on day 65? We don't want to keep going to the shelter for the next 300 days just because our original plan was to go for a whole year.

In our code, when we want to stop a loop from continuing to execute even though the original stopping condition we wrote for our loop hasn't been met, we can use the keyword break.

const rapperArray = ["Lil' Kim", "Jay-Z", "Notorious B.I.G.", "Tupac"];

// Write you code below

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

console.log(rapperArray[i]);

if (rapperArray[i] === 'Notorious B.I.G.'){

break;

}

}

console.log("And if you don't know, now you know."

[**JavaScript: Iterators**](https://www.codecademy.com/learn/paths/web-development)

**Introduction**

If we told you to "count to three," we would expect you to say or think the numbers one, two and three. With programming, we’re faced with needing to be more explicit with our directions to the computer. Here's how we might tell the computer to "count to three":

for (let i = 1; i<=3; i++) {

console.log(i)

}

**HIGHER-ORDER FUNCTIONS**

Instead of listing all those details, we can say, "We baked a cake," and still impart all that meaning to you. Not required abstract detail for humans.

In programming, we can accomplish “abstraction” by writing functions. In addition to allowing us to reuse our code, functions help to make clear, readable programs.

Higher-order functions are functions that accept other functions as arguments and/or return functions as output.

Example : "We hosted a birthday party" is an abstraction that may build on the abstraction "We made a cake."

**Functions as Data**

Here, we have an annoyingly long function name that hurts the readability of any code in which it's used. This function going to call repeatly.

const announceThatIAmDoingImportantWork = () => {

console.log("I’m doing very important work!");

};

We can re-assign the function to a variable with a suitably short name:

const busy = announceThatIAmDoingImportantWork;

busy(); // This function call barely takes any space!

busy is a variable that holds a **reference** to our original function.

Notice how we assign announceThatIAmDoingImportantWork without parentheses as the value to the busy variable.

Since functions are a type of object, they have properties such as .length and .name and methods such as .toString(). You can see more about the methods and properties of functions

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function>

const checkThatTwoPlusTwoEqualsFourAMillionTimes = () => {

for(let i = 1; i <= 1000000; i++) {

if ( (2 + 2) != 4) {

console.log('Something has gone very wrong :( ');

}

}

}

// change name of a function and dont use () with orignal function name.

let is2p2= checkThatTwoPlusTwoEqualsFourAMillionTimes;

// invoke

is2p2();

// if we forgot the original name of our function

//use of property

console.log(is2p2.name);

**Functions as Parameters**

We can also pass functions (into other functions) as parameters. A *higher-order function* is a function that either accepts functions as parameters, returns a function, or both! We call the functions that get passed in as parameters and invoked *callback functions* because they get called during the execution of the higher-order function.

When we pass a function in as an argument to another function, we don't invoke it. Invoking the function would evaluate to the return value of that function call. With callbacks, we pass in the function itself by typing the function name *without* the parentheses (that would evaluate to the result of calling the function):

const timeFuncRuntime = funcParameter => {

let t1 = Date.now();

funcParameter();

let t2 = Date.now();

return t2 - t1;

}

const addOneToOne = () => 1 + 1;

timeFuncRuntime(addOneToOne);

We wrote a higher-order function, timeFuncRuntime(). It takes in a function as an argument, saves a starting time, invokes the callback function, records the time after the function was called, and returns the time the function took to run by subtracting the starting time from the ending time.

This higher-order function could be used with any callback function which makes it a potentially powerful piece of code.

We then invoked timeFuncRuntime() first with the addOneToOne() function - note how we passed in addOneToOne and did not invoke it.

timeFuncRuntime(() => {

for (let i = 10; i>0; i--){

console.log(i);

}

});

In this example, we invoked timeFuncRuntime()with an anonymous function that counts backwards from 10. Anonymous functions can be arguments too!

const checkThatTwoPlusTwoEqualsFourAMillionTimes = () => {

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

if ( (2 + 2) !=4 ) {

console.log('Something has gone very wrong :( ');

}

else {

console.log("running function 1")

}

}

};

const addTwo = num => num + 2;

const timeFuncRuntime = funcParameter => {

let t1 = Date.now();

funcParameter();

let t2 = Date.now();

console.log("hey")

return t2 - t1;

};

// Write your code below

const time2p2 = timeFuncRuntime(checkThatTwoPlusTwoEqualsFourAMillionTimes);

const checkConsistentOutput = (func, val) => {

let firstTry = func(val);// addTwo(val+2)

let secondTry = func(val);

if (firstTry === secondTry) {

return firstTry

} else {

return 'This function returned inconsistent results'

}

};

console.log(checkConsistentOutput(addTwo, 10))

**Introduction to Iterators**

The built-in JavaScript array methods that help us iterate are called *iteration methods*, at times referred to as *iterators*. Iterators are methods called on arrays to manipulate elements and return values.

const artists = ['Picasso', 'Kahlo', 'Matisse', 'Utamaro'];

artists.forEach(artist => {

console.log(artist + ' is one of my favorite artists.');

});

const numbers = [1, 2, 3, 4, 5];

const squareNumbers = numbers.map(number => {

return number \* number;

});

console.log(squareNumbers);

const things = ['desk', 'chair', 5, 'backpack', 3.14, 100];

const onlyNumbers = things.filter(thing => {

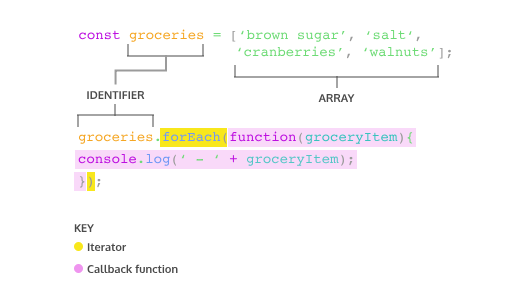
return typeof thing === 'number';

});

console.log(onlyNumbers);

**The .forEach() Method**

Aptly named, .forEach() will execute the same code for each element of an array.



Example:

const fruits = ['mango', 'papaya', 'pineapple', 'apple'];

// Iterate over fruits below

/\* way 1:---------------------------------

function printFruits(element){

console.log("I want to eat a "+element)

}

fruits.forEach(printFruits);

-------------------------------------------

\*/

/\*

Way 2-------------------------------------

fruits.forEach(function(printFruits){

console.log("i want to eat a "+printFruits)

})

------------------------------------------

\*/

//way-3

fruits.forEach(printFruits => console.log("i want to eat a "+printFruits));

**The .map() Method**

The second iterator we're going to cover is .map(). When .map() is called on an array, it takes an argument of a callback function and returns a new array! Take a look at an example of calling .map():

const numbers = [1, 2, 3, 4, 5];

const bigNumbers = numbers.map(number => {

return number \* 10;

});

.map() works in a similar manner to .forEach()— the major difference is that .map() returns a new array.

If we take a look at numbers and bigNumbers:

console.log(numbers); // Output: [1, 2, 3, 4, 5]

console.log(bigNumbers); // Output: [10, 20, 30, 40, 50]

Notice that the elements in numbers were not altered and bigNumbers is a new array.

//Program -1

const animals = ['Hen', 'elephant', 'llama', 'leopard', 'ostrich', 'Whale', 'octopus', 'rabbit', 'lion', 'dog'];

// WAY -1

const secretMessage = animals.map(animals => {

return animals[0]

});

console.log(secretMessage.join(''));

//Output: HelloWorld is a Secret message

//-------------------------------------------------------------------

//Program -2

// WAY -1

const bigNumbers = [100, 200, 300, 400, 500];

const smallNumbers = bigNumbers.map(bigNumbers => {

return bigNumbers /100

});

console.log(smallNumbers.join(''));

// Create the smallNumbers array below

//Output: 12345

//---------------------------------------------------------------------

//WAY -2 [DIRECT ACCESS]

const mapWay2 = [1000, 2000, 3000, 4000, 5000].map(function(elements){

return elements /100;

});

console.log(mapWay2);

//---------------------------------------------------------------------

//WAY -3 [DIRECT ACCESS]

const mapWay3 = [1000, 2000, 3000, 4000, 5000].map(elements=>elements /10);

console.log(mapWay3);

**The .filter() Method**

Like .map(), .filter() returns a new array.

However, .filter() returns an array of elements after filtering out certain elements from the original array. The callback function for the .filter()method should return true or false depending on the element that is passed to it.

const words = ['chair', 'music', 'pillow', 'brick', 'pen', 'door'];

const shortWords = words.filter(word => {

return word.length < 6;

});

* word.length < 6; is the condition in the callback function. Any word from the wordsarray that has fewer than 6 characters will be added to the shortWords array.

Let's also check the values of words and shortWords:

console.log(words); // Output: ['chair', 'music', 'pillow', 'brick', 'pen', 'door'];

console.log(shortWords); // Output: ['chair', 'music', 'brick', 'pen', 'door']

Example :

const randomNumbers = [375, 200, 3.14, 7, 13, 852];

//Way -1

const smallNumbers = randomNumbers.filter(function(randomNumbers){

return randomNumbers < 250;

})

console.log(smallNumbers)

// Output : [ 200, 3.14, 7, 13 ]

//----------------------------------------------------------------------------------------------

const randomNumbers2 = [3750, 2000, 3.140, 70, 130, 8520];

//Way -2

const filteredNums2 = randomNumbers2.filter(randomNumbers2 => randomNumbers2 < 2500);

console.log(filteredNums2);

//Output : [ 2000, 3.14, 70, 130 ]

//----------------------------------------------------------------------------------------------

const randomNumbers3 = [3750, 2000, 3.140, 70, 130, 8520];

//Way-3

const Way3 = randomNumbers3.filter(randomNumbers3 => {

return randomNumbers3 < 250

});

console.log(Way3);

//Output : [ 3.14, 70, 130 ]

//----------------------------------------------------------------------------------------------

//Way -4 [Direct Access]

const Way4 = [37500, 20000, 3.14000, 700000, 130000, 85200].filter(function(elements){

return elements < 10000;

});

console.log(Way4);

//Output :[ 3.14 ]

//----------------------------------------------------------------------------------------------

//Way-5 [Direct Access]

const filterWay5 = [1000, 2000, 3000, 4000, 5000].filter(elements=>elements <= 2000);

console.log(filterWay5);

// Output : [ 1000, 2000 ]

//----------------------------------------------------------------------------------------------

//Way -6

const numbers1 = [10, 20, 30, 40, 50];

const filteredNums = numbers1.filter(function(number1){

if (number1 < 25) {

return true;

}

})

console.log(numbers1);

//----------------------------------------------------------------------------------------------

// WORK WITH STRINGS

const favoriteWords = ['nostalgia', 'hyperbole', 'fervent', 'esoteric', 'serene'];

const longFavoriteWords = favoriteWords.filter(function(favoriteWords){

return favoriteWords.length > 7;

})

console.log(longFavoriteWords)

# The .findIndex() Method

Calling .findIndex() on an array will return the index of the first element that evaluates to true in the callback function.

const jumbledNums = [123, 25, 78, 5, 9];

const lessThanTen = jumbledNums.findIndex(num => {

return num < 10;

});

console.log(lessThanTen); // Output: 3

If we check what element has index of 3:

console.log(jumbledNums[3]); // Output: 5

If there isn't a single element in the array that satisfies the condition in the callback, then .findIndex() will return -1.

const greaterThan1000 = jumbledNums.findIndex(num => {

return num > 1000;

});

console.log(greaterThan1000); // Output: -1

Example : 2

const animals = ['hippo', 'tiger', 'lion', 'seal', 'cheetah', 'monkey', 'salamander', 'elephant'];

const foundAnimal = animals.findIndex(animal => {

return animal === 'elephant';

});

const startsWithS = animals.findIndex(animal => {

return animal[0] === 's' ? true : false;

});

console.log(foundAnimal);

console.log(startsWithS);

# The .reduce() Method

The .reduce() method returns a single value after iterating through the elements of an array, thereby reducing the array. Take a look at the example below:

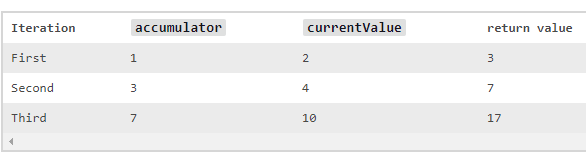
const numbers = [1, 2, 4, 10];

const summedNums = numbers.reduce((accumulator, currentValue) => {

return accumulator + currentValue

})

console.log(summedNums) // Output: 17



* The callback function has two parameters, accumulator and currentValue. The value of accumulator starts off as the value of the first element in the array and the currentValue starts as the second element. To see the value of accumulator and currentValue change, review the chart above.

The .reduce() method can also take an optional second parameter to set an initial value for accumulator (remember, the first argument is the callback function!). For instance:

//accumulator is a callback fun

const numbers = [1, 2, 4, 10];

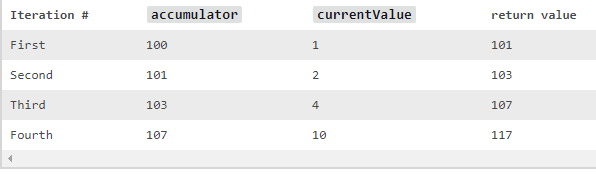
const summedNums = numbers.reduce((accumulator, currentValue) => {

return accumulator + currentValue

}, 100) // <- Second argument for .reduce()

console.log(summedNums); // Output: 117

Here's an updated chart that accounts for the second argument of 100:



Example :

const newNumbers = [1, 3, 5, 7];

const newSum = newNumbers.reduce(function(accumulator, currentValue) {

console.log('The value of accumulator: ', accumulator);

console.log('The value of currentValue: ', currentValue);

return accumulator + currentValue;

}, 10); // <- Second argument for .reduce()

console.log("Total sum :"+newSum)

Output:

The value of accumulator: 10

The value of currentValue: 1

The value of accumulator: 11

The value of currentValue: 3

The value of accumulator: 14

The value of currentValue: 5

The value of accumulator: 19

The value of currentValue: 7

Total sum :26

***-->Every()***

***Every element of an array should satify the condition then it returns true else false.***

function isBelowThreshold(currentValue) {

return currentValue < 40;

}

var array1 = [1, 30, 39, 29, 10, 13];

console.log(array1.every(isBelowThreshold));

// expected output: true

**Array.some()**

**If one element of an array satisfy the condition. it returns true.**

function isBiggerThan10(element, index, array) {

return element > 10;

}

[2, 5, 8, 1, 4].some(isBiggerThan10); // false

[12, 5, 8, 1, 4].some(isBiggerThan10); // true

Find more:---------

"https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/Array#Iteration\_methods"

Example:

const cities = ['Erlando', 'Aubai', 'Rdinburgh', 'Nhennai', 'Cccra', 'Oenver', 'Oskisehir', 'Dedellin', 'Eokohama'];

const nums = [1, 50, 75, 200, 350, 525, 1000];

// Choose a method that will return undefined

cities.forEach(city => console.log('Have you visited ' + city + '?'));

// Choose a method that will return a new array

const longCities = cities.filter(city => city.length > 7);

console.log(longCities);

// Choose a method that will return a single value

const word = cities.reduce((acc, currVal) => {

return acc + currVal[0]

}, "L");

console.log(word)

// Choose a method that will return a new array

const smallerNums = nums.map(num => num - 5);

console.log(smallerNums);

// Choose a method that will return a boolean value

const a = nums.some(num => num < 5);

console.log(a)

// Choose a method that will return a boolean value

const b = nums.every(num => num < 5);

console.log(b)

Project : [**JavaScript: Iterators**](https://www.codecademy.com/learn/paths/web-development)

Manipulation In string Paragraph

let story = 'Last weekend, I took literally the most beautiful bike ride of my life. The route is called "The 9W to Nyack" and it actually stretches all the way from Riverside Park in Manhattan to South Nyack, New Jersey. It\'s really an adventure from beginning to end! It is a 48 mile loop and it basically took me an entire day. I stopped at Riverbank State Park to take some extremely artsy photos. It was a short stop, though, because I had a really long way left to go. After a quick photo op at the very popular Little Red Lighthouse, I began my trek across the George Washington Bridge into New Jersey. The GW is actually very long - 4,760 feet! I was already very tired by the time I got to the other side. An hour later, I reached Greenbrook Nature Sanctuary, an extremely beautiful park along the coast of the Hudson. Something that was very surprising to me was that near the end of the route you actually cross back into New York! At this point, you are very close to the end.';

//Split words in an string

let split = story.split(' ');

//console.log(split);

let FindLength = story.length;

console.log("total Orignal Words",FindLength);

//remove these words used INCLUDE METHOD with ! operator

let unnecessaryWords = ['extremely', 'literally', 'actually' ];

const betterWords = split.filter(word => {

return !unnecessaryWords.includes(word);

});

//console.log(betterWords);

//now check the length of words

console.log("After Alter ->Word counts :",betterWords.length);

let overusedWords = ['really', 'very', 'basically'];

let reallyCount=0;

let veryCount=0;

let basicallyCount=0;

for(word of split){

if(word==="really")

{

reallyCount +=1;

}

else if(word==="very")

{

veryCount +=1;

}

else if(word==="basically")

{

basicallyCount +=1;

}

}

console.log("really count :" + reallyCount)

console.log("very count :" + veryCount)

console.log("basically count :" + basicallyCount)

let sentenceCount=0;

for(word of split){

if (word[word.length - 1]==='.' || word[word.length - 1]==='!')

{

sentenceCount += 1;

}

}

console.log("sentences count: ",sentenceCount);

console.log(betterWords.join(' '))

# Introduction to Objects

There are only seven fundamental data types in JavaScript, and six of those are the primitive data types: string, number, boolean, null, undefined, and symbol.With the seventh type, objects

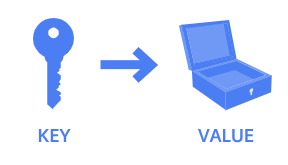
JavaScript objects are containers storing related data and functionality, but that deceptively simple task is extremely powerful in practice.

# Creating Object Literals

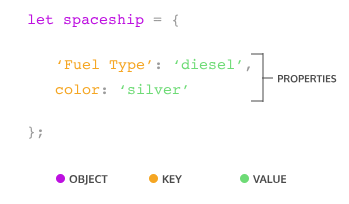
object literal created:

let spaceship = {}; // spaceship is an empty object

We fill an object with unordered data. This data is organized into key(variable-name point location that holds value)-value pairs.



A key's value can be of any data type in the language including functions or other objects.



// An object literal with two key-value pairs

let spaceship = {

'Fuel Type': 'diesel', // define properties

color: 'silver' // fuel type, has quotation marks because it contains a space character.

};

# Accessing Properties

There are two ways we can access an object's property.

first way— dot notation, .

let spaceship = {

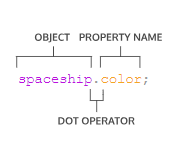
homePlanet: 'Earth',

color: 'silver'

};

spaceship.homePlanet; // Returns 'Earth',

spaceship.color; // Returns 'silver',



**Note** :If we try to access a property that does not exist on that object, `undefined` will be returned.

let spaceship = {

homePlanet: 'Earth',

color: 'silver',

'Fuel Type': 'Turbo Fuel', // accessing problem can be solved through []

numCrew: 5,

flightPath: ['Venus', 'Mars', 'Saturn']

};

// access values of an object

let crewCount =spaceship.numCrew;

let planetArray =spaceship.flightPath ;

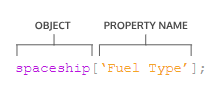
# Bracket Notation

The second way [ ].

You've used bracket notation when indexing an array:

['A', 'B', 'C'][0]; // Returns 'A'

To use bracket notation to access an object's property, we pass in the property name (key) as a string.



We \*must\* use bracket notation when accessing keys that have numbers, spaces, or special characters in them. Without bracket notation in these situations, our code would throw an error.

let spaceship = {

'Fuel Type': 'Turbo Fuel',

'Active Duty': true,

homePlanet: 'Earth',

numCrew: 5

};

spaceship['Active Duty']; // Returns true

spaceship['Fuel Type']; // Returns 'Turbo Fuel'

spaceship['numCrew']; // Returns 5

spaceship['!!!!!!!!!!!!!!!']; // Returns undefined

With bracket notation you can also use a variable inside the brackets to select the keys of an object. This can be especially helpful when working with functions:

Syntax:

let returnAnyProp = (objectName, propName) => objectName[propName];

returnAnyProp(spaceship, 'homePlanet'); // Returns 'Earth'

Example :

let spaceship = {

'Fuel Type': 'Turbo Fuel',

'Active Duty': true,

homePlanet: 'Earth',

numCrew: 5

};

//Way -1

let abc = function (spaceship, propName){

return spaceship[propName];

}

console.log(abc(spaceship,'homePlanet'));

console.log(abc(spaceship,'Active Duty'));

console.log(abc(spaceship,'numCrew'));

console.log(abc(spaceship,'Fuel Type'));

// WAY -2

//let returnAnyProp = (spaceship, propName) => spaceship[propName];

//console.log(returnAnyProp(spaceship, 'numCrew'))

//console.log(returnAnyProp(spaceship, 'homePlanet'));

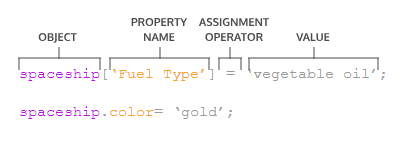
//console.log(returnAnyProp(spaceship, 'Active Duty'))

//console.log(returnAnyProp(spaceship, 'Fuel Type'));

# Property Assignment

There is no need to stuck with all the properties we wrote. Objects are mutable meaning we can update them after we create them!

We can use either dot notation, ., or bracket notation, [], and the assignment operator, = to add new key-value pairs to an object or change an existing property.



One of two things can happen with property assignment:

* If the property already exists on the object, whatever value it held before will be replaced with the newly assigned value.
* If there was no property with that name, a new property will be added to the object.

It's important to know that although we can't reassign an object declared with const, we can still mutate it, meaning we can add new properties and change the properties that are there.

const spaceship = {type: 'shuttle'};

spaceship = {type: 'alien'}; // TypeError: Assignment to constant variable if we do same thing with let then it works.

spaceship.type = 'alien'; // Changes the value of the type property

spaceship.speed = 'Mach 5'; // Creates a new key of 'speed' with a value of 'Mach 5'

// Assign property with use of brackets.

objectName['Property Name'] = 'New Property Value';

You can delete a property from an object with the deleteoperator.

const spaceship = {

'Fuel Type': 'Turbo Fuel',

homePlanet: 'Earth',

mission: 'Explore the universe'

};

delete spaceship.mission; // Removes the mission property

delete spaceship['mission']

# Methods

When the data stored on an object is a function we call that a method.

Do object methods seem familiar? That’s because you've been using them all along! For example console is a global javascript object and .log()is a method on that object. Math is also a global javascript object and .floor() is a method on it.

const alienShip = {

invade: function () {

console.log('Hello! We have come to dominate your planet. Instead of Earth, it shall be called New Xaculon.')

}

};

Way - 2

const alienShip = {

// With the new method syntax introduced in ES6 we can omit the colon and the function keyword.

invade () {

console.log('Hello! We have come to dominate your planet. Instead of Earth, it shall be called New Xaculon.')

}

};

alienShip.invade(); // Prints 'Hello! We have come to dominate your planet. Instead of Earth, it shall be called New Xaculon.'

Example :

let retreatMessage = 'We no longer wish to conquer your planet. It is full of dogs, which we do not care for.';

// Write your code below

let alienShip = {

passenger:null,

// create function in an object

retreat() {

console.log(retreatMessage);

},

takeOff () {

console.log('Spim... Borp... Glix... Blastoff!')

},// seperated by comma

homePlanet: 'Earth',

numCrew: 5,

}

// access those functions which is declare in a object

alienShip.retreat();

alienShip.takeOff();

let a = alienShip.homePlanet;

let b = alienShip['numCrew'];

console.log(a,b)

// add new elements in an object

alienShip.speed = 'Mach 5';

alienShip['bike'] = 'pulser';

// accessing new elements

let c = alienShip.speed;

let d = alienShip['bike'];

console.log(c,d)

// just an example \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

spaceship.passengers = [{name: 'Space Dog'},{name: 'Space bag'}];

# Nested Objects

In application code, objects are often nested— an object might have another object as a property which in turn could have a property that's an array of even more objects!

const spaceship =

{

telescope:

{

yearBuilt: 2018,

model: '91031-XLT',

focalLength: 2032

},

crew:

{

captain:

{

name: 'Sandra',

degree: 'Computer Engineering',

encourageTeam() { console.log('We got this!') }

}

},

engine:

{

model: 'Nimbus2000'

},

nanoelectronics:

{

computer:

{

terabytes: 100,

monitors: 'HD'

},

'back-up':

{

battery: 'Lithium',

terabytes: 50

}

}

};

We can chain operators to access nested properties.

spaceship.nanoelectronics['back-up'].battery; // Returns 'Lithium'

// just an example

let capFave = spaceship.crew.captain['favorite foods'][0];

# Pass By Reference

Objects are passed by reference. This means when we pass a variable assigned to an object into a function as an argument, the computer interprets the parameter name as pointing to the space in memory holding that object. As a result, functions which change object properties actually mutate the object permanently (even when the object is assigned to a const variable).

const spaceship = {

homePlanet : 'Earth',

color : 'silver'

};

let paintIt = obj => {

obj.color = 'glorious gold'

obj.disabled = true; // new element };

paintIt(spaceship);

spaceship.color // Returns 'glorious gold' true

Our function paintIt() permanently changed the color of our spaceship object. However, reassignment of the spaceship variable wouldn't work in the same way:

let spaceship = {

homePlanet : 'Earth',

color : 'red'

};

let tryReassignment = obj => {

obj = {

identified : false,

'transport type' : 'flying'

}

console.log(obj) // Prints {'identified': false, 'transport type': 'flying'}

};

tryReassignment(spaceship) // The attempt at reassignment does not work.

spaceship // Still returns {homePlanet : 'Earth', color : 'red'};

spaceship = {

identified : false,

'transport type': 'flying'

}; // Regular reassignment still works.

Let's look at what happened in the code example:

* We declared this spaceship object with let. This allowed us to reassign it to a new object with identified and 'transport type'properties with no problems.
* When we tried the same thing using a function designed to reassign the object passed into it, the reassignment didn't stick (even though calling console.log() on the object produced the expected result).
* When we passed spaceship into that function, obj became a reference to the memory location of the spaceship object, but not to the spaceship variable. This is because the obj parameter of the tryReassignment()function is a variable in its own right. The body of tryReassignment() has no knowledge of the spaceship variable at all!
* When we did the reassignment in the body of tryReassignment(), the obj variable came to refer to the memory location of the object {'identified' : false, 'transport type' : 'flying'}, while the spaceship variable was completely unchanged from its earlier value.

# Looping Through Objects

Loops are programming tools that repeat a block of code until a condition is met. We learned how to iterate through arrays using their numerical indexing, but the key-value pairs in objects aren't ordered!

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in

Syntax:-

for (let variableName in outerObject.innerObject) {

console.log(`${outerObject.innerObject[variableName].propertyName}: ${outerObject.innerObject[variableName].differentPropertyName}`)

};

for...in will execute a given block of code for each property in an object.

let spaceship = {

crew: {

captain: {

name: 'Lily',

degree: 'Computer Engineering',

cheerTeam() { console.log('You got this!') }

},

'chief officer': {

name: 'Dan',

degree: 'Aerospace Engineering',

agree() { console.log('I agree, captain!') }

},

medic: {

name: 'Clementine',

degree: 'Physics',

announce() { console.log(`Jets on!`) } },

translator: {

name: 'Shauna',

degree: 'Conservation Science',

powerFuel() { console.log('The tank is full!') }

}

}

};

// for...in

for (let crewMember in spaceship.crew) {

console.log(`${crewMember}: ${spaceship.crew[crewMember].name}`)

};

for (let crewMember in spaceship.crew) {

console.log(`${spaceship.crew[crewMember].name}: ${spaceship.crew[crewMember].degree}`)

};

Our for...in will iterate through each element of the spaceship.crew object. In each iteration, the variable crewMember is set to one of spaceship.crew's keys, enabling us to log a list of crew members' role and name.

const car = {

numDoors: 4,

isDirty: true,

color: 'red'

}

for (let key in car) {

console.log(key)

}

Output:

numDoors

isDirty

color

# Advanced Objects Introduction

Remember, objects in JavaScript are containers that store data and functionality.

Advance Concepts Topics:

* how to use the this keyword.
* conveying privacy in JavaScript methods.
* defining getters and setters in objects.
* creating factory functions.
* using destructuring techniques.

# The this Keyword

Objects are collections of related data and functionality. We store that functionality in methods on our objects:

const goat = {

dietType: 'herbivore',

makeSound() {

console.log('baaa');

}

};

In our goat object we have a .makeSound()method. We can invoke the .makeSound() method on goat.

goat.makeSound(); // Prints baaa

// Everything seems to be working fine.

What if we wanted to add a new method to our goat object called .diet() that prints the goat's dietType?

const goat = {

dietType: 'herbivore',

makeSound() {

console.log('baaa');

},

diet() {

console.log(dietType);

}

};

goat.diet();

// Output will be "ReferenceError: dietType is not defined"

That's strange, why is dietType not defined even though it's a property of goat? That's because inside the scope of the .diet() method, we don't automatically have access to other properties of the goat object.

***this keyword comes to the rescue.***

const robot = {

model:'1E78V2',

energyLevel:100,

provideInfo(){

return `I am ${this.model} and my current energy level is ${this.energyLevel}`

}

};

console.log(robot.provideInfo());

The this keyword references the calling objectwhich provides access to the calling object's properties.

Here, we can use that diet() function but, with the use of goat object.

const robot = {

model:'1E78V2',

energyLevel:100,

provideInfo(){

return `I am ${robot.model} and my current energy level is ${robot.energyLevel}` // see the DIFFERENCE

}

};

console.log(robot.provideInfo());

**Arrow Functions and this**

We saw in the previous exercise that for a method, the calling object is the object the method belongs to. If we use the this keyword in a method then the value of this is the calling object. However, it becomes a bit more complicated when we start using arrow functions for methods.

const goat = {

dietType: 'herbivore',

makeSound() {

console.log('baaa');

},

diet: () => {

console.log(this.dietType);

} // (THIS KEYWORD) NOT WORK WITH ARROW FUNCTONS

};

goat.diet(); // Prints undefined

const goat = {

dietType: 'herbivore',

makeSound() {

console.log('baaa');

},

diet: () => {

console.log(robot.dietType);

} // NOW WORKS

};

goat.diet(); // Output : Energy is currently at 100%.

Arrow functions inherently bind, or tie, an already defined this value to the function itself that is NOT the calling object. To read more about either arrow functions or the global object check out the

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions>

<https://developer.mozilla.org/en-US/docs/Glossary/Global_object>

**Privacy**

When discussing *privacy* in objects, we define it as the idea that only certain properties should be mutable or able to change in value.

One common convention is to place an underscore \_ before the name of a property to mean that the property should not be altered. Here's an example of using \_ to prepend a property.

const bankAccount = {

\_amount: 1000

}

Even so, it is still possible to reassign \_amount:

bankAccount.\_amount = 1000000;

const robot = {

\_energyLevel: 'high',

recharge(){

this.\_energyLevel += 30;

console.log(`Recharged! Energy is currently at ${this.\_energyLevel}%.`)

}

};

robot.recharge();

Output:

Recharged! Energy is currently at high30%.

// This is known as a side-effect of type-coercion.

The use of methods called getters and setters. Both methods are used to respect the intention of properties prepended, or began, with \_. Getters can return the value of internal properties and setters can safely reassign property values.

**Getters**

*Getters* are methods that get and return the internal properties of an object.

Name of the getter and setter function should not be same as properties name.

const person = {

\_firstName: 'John',

\_lastName: 'Doe',

get fullName() {

if (this.\_firstName && this.\_lastName){

return `${this.\_firstName} ${this.\_lastName}`;

} else {

return 'Missing a first name or a last name.';

}

}

}

// To call the getter method:

person.fullName; // 'John Doe'

Advantages of using a getter method:

* Getters can perform an action on the data when getting a property.
* Getters can return different values using conditionals.
* In a getter, we can access the properties of the calling object using this.
* The functionality of our code is easier for other developers to understand.

Another thing to keep in mind when using getter (and setter) methods is that properties cannot share the same name as the getter/setter function. If we do so, then calling the method will result in an infinite call stack error. One workaround is to add an underscore before the property name like we did in the example above.

Example:

const robot = {

\_model: '1E78V2',

\_energyLevel: 100,

get energyLevel(){

if(typeof this.\_energyLevel == 'number' )

{

return `My current energy level is ${this.\_energyLevel}`

}

else{

return 'System malfunction: cannot retrieve energy level'

}

}

};

console.log(robot.energyLevel)

**Setters**

Along with getter methods, we can also create *setter* methods which reassign values of existing properties within an object.

When You need to create setter methods, because when you want anyone to change the values saved to these keys.

const person = {

\_age: 37,

set age(newAge){

if (typeof newAge === 'number'){

this.\_age = newAge;

} else {

console.log('You must assign a number to age');

}

}

};

//Then to use the setter method:

person.age = 40;

console.log(person.\_age); // Logs: 40

person.age = '40'; // Logs: You must assign a number to age

Setter methods like age do not need to be called with a set of parentheses. Syntactically, it looks like we're reassigning the value of a property.

**Factory Functions**

We create objects individually, but there are times where we want to create many instances of an object quickly. Here's where *factory functions* come in.

A real world factory manufactures multiple copies of an item quickly and on a massive scale. Factory functions can also have parameters allowing us to customize the object that gets returned.

const monsterFactory = (name, age, energySource, catchPhrase) => {

return {

name: name,

age: age,

energySource: energySource,

scare() {

console.log(catchPhrase);

}

}

};

In the monsterFactory function above, it has four parameters and returns an object that has the properties: name, age, energySource, and scare(). To make an object that represents a specific monster like a ghost. we can call monsterFactory with the necessary arguments and assign the return value to a variable:

const ghost = monsterFactory('Ghouly', 251, 'ectoplasm', 'BOO!');

ghost.scare(); // 'BOO!'

Now we have a ghost object as a result of calling monsterFactory() with the needed arguments. With monsterFactory in place, we don't have to create an object literal every time we need a new monster. Instead, we can invoke the monsterFactory function with the necessary arguments to  make a monster for us!

Example :

const robotFactory = (model,mobile) => {

return {

model: model,

mobile: mobile,

beep(){

console.log("Beep Boop")

}

}

};

const tinCan=robotFactory('P-500',true);

tinCan.beep();

console.log(tinCan.model);

console.log(tinCan.mobile);

**Property Value Shorthand**

ES6 introduced some new shortcuts for assigning properties to variables known as *destructuring*.

const monsterFactory = (name, age) => {

return {

name: name,

age: age

}

};

Imagine if we had to include more properties, that process would quickly become tedious! But we can use a destructuring technique, called property value shorthand, to save ourselves some keystrokes.

const monsterFactory = (name, age) => {

return {

name,

age

// We had to assign each property a key and value even though the key name was the same as the parameter name we assigned to it

}

};

Example:

function robotFactory(model, mobile){

return {

model,

mobile,

beep() {

console.log('Beep Boop');

}

}

}

// To check that the property value shorthand technique worked:

const newRobot = robotFactory('P-501', false)

console.log(newRobot.model)

console.log(newRobot.mobile)

newRobot.beep();

**Destructured Assignment**

We often want to extract key-value pairs from objects and save them as properties. Take for example the following object:

const vampire = {

name: 'Dracula',

residence: 'Transylvania',

preferences: {

day: 'stay inside',

night: 'satisfy appetite'

}

};

If we wanted to extract the residence property as a variable, we could using the following code:

const residence = vampire.residence;

console.log(residence); // Prints 'Transylvania'

However, we can also take advantage of a destructuring technique called destructured assignment to save ourselves some keystrokes. In destructured assignment we create a variable with the name of an object's key that is wrapped in curly braces { } and assign to it the object. Take a look at the example below:

const { residence } = vampire;

console.log(residence); // Prints 'Transylvania'

Look back at the vampire object's properties in the first code example. Then, in the example above, we declare a new variable residence that extracts the value of the residence property of vampire. When we log the value of residence to the console, 'Transylvania' is printed.

We can even use destructured assignment to grab nested properties of an object:

const { day } = vampire.preferences;

console.log(day); // Prints 'stay inside'

Example:

const robot = {

model: '1E78V2',

energyLevel: 100,

functionality: {

beep() {

console.log('Beep Boop');

},

fireLaser() {

console.log('Pew Pew');

},

}

};

const {functionality}= robot;

functionality.beep();

**Built-in Object Methods**

**Object.keys()**

The **Object.keys()** method returns an array of a given object's own property **names**, in the same order as we get with a normal loop.

const robot = {

    model: 'SAL-1000',

mobile: true,

sentient: false,

armor: 'Steel-plated',

energyLevel: 75

};

// What is missing in the following method call?

const robotKeys = Object.keys(robot);

console.log(robotKeys);

Output:

[ 'model', 'mobile', 'sentient', 'armor', 'energyLevel' ]

**Object.entries()**

The **Object.entries()** method returns an array of a given object's own enumerable property [key, value] pairs, in the same order as that provided by a [for...in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in) loop (the difference being that a for-in loop enumerates properties in the prototype chain as well) . The order of the array returned by **Object.entries()** does not depend on how an object is defined. If there is a need for certain ordering then the array needs to be sorted first like Object.entries(obj).sort((a, b) => a[0] - b[0]);.

const robot = {

    model: 'SAL-1000',

mobile: true,

sentient: false,

armor: 'Steel-plated',

energyLevel: 75

};

const robotEntries = Object.entries(robot);

console.log(robotEntries);

Output:

[ [ 'model', 'SAL-1000' ],

[ 'mobile', true ],

[ 'sentient', false ],

[ 'armor', 'Steel-plated' ],

[ 'energyLevel', 75 ] ]

**Object.assign()**

The **Object.assign()** method is used to copy the values of all enumerable own properties from one or more source objects to a target object. It will return the target object.

const robot = {

    model: 'SAL-1000',

mobile: true,

sentient: false,

armor: 'Steel-plated',

energyLevel: 75

};

const newRobot = Object.assign({laserBlaster: true, voiceRecognition: true}, robot);

console.log(newRobot);

Output:

// assign() Object methods can be used to copy all of the properties of an object into a new object

{ laserBlaster: true,

voiceRecognition: true,

model: 'SAL-1000',

mobile: true,

sentient: false,

armor: 'Steel-plated',

energyLevel: 75 }

For more:

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object#Methods_of_the_Object_constructor>

Project -1

To Understand Object inserting and fetching elements

const team = {

\_players: [

{

firstName: 'shubham',

lastName: 'mathur',

age: 23

},

{

firstName: 'sonu',

lastName: 'balaji',

age: 31

},

{

firstName: 'kaku',

lastName: 'Sanchez',

age: 31

}

],

\_games: [

{

opponent: 'sak',

teamPoints: 42,

opponentPoints: 27

},

{

opponent: 'duk',

teamPoints: 4,

opponentPoints: 7

},

{

opponent: 'muk',

teamPoints: 2,

opponentPoints: 2

}

],

get players(){

return this.team.\_players;

},

get games(){

return this.team.\_games;

},

/\*name should not be same in getter and setter \*/

// INSERT NEW ELEMENTS IN AN ARRAY OF OBJECT USING METHODS

addPlayer(firstName, lastName, age){

let player ={

firstName: firstName,

lastName: lastName,

age: age

}

this.\_players.push(player); // here may be mistake

},

addGame(opp, myPts, oppPts) {

const game = {

opponent: opp,

points: myPts,

opponentPoints: oppPts

};

this.\_games.push(game); // error can come

}

}

// console all players

team.addPlayer('RANI', 'MUKHARJII', 28);

team.addPlayer('prachi', 'sharma', 12);

team.addPlayer('priyanshi', 'jain', 23);

console.log(team.\_players);

// console all game oppoint

team.addGame('kola', 120, 33);

team.addGame('dudu', 32, 22);

team.addGame('tuimur', 33, 22);

console.log(team.\_games);

Example 2 of Project 1(fetching data from getter method) :

const team = {

\_players: [

{

firstName: 'shubham',

lastName: 'mathur',

age: 23

},

{

firstName: 'sonu',

lastName: 'balaji',

age: 31

},

{

firstName: 'kaku',

lastName: 'Sanchez',

age: 31

}

],

\_games: [

{

opponent: 'sak',

teamPoints: 42,

opponentPoints: 27

},

{

opponent: 'duk',

teamPoints: 4,

opponentPoints: 7

},

{

opponent: 'muk',

teamPoints: 2,

opponentPoints: 2

}

],

get players(){

if (this.\_players && this.\_games){

return this.\_players

} else {

return 'Missing a first name or a last name.';

}

}}

console.log(team.players);