**JS ES6**

**Compare Scopes of the var and let Keywords**

If you are unfamiliar with let, check out [this challenge](https://www.freecodecamp.org/learn/javascript-algorithms-and-data-structures/basic-javascript/explore-differences-between-the-var-and-let-keywords).

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

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

For example:

var numArray = [];

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

numArray.push(i);

}

console.log(numArray);

console.log(i);

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

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

var numArray = [];

var i;

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

numArray.push(i);

}

console.log(numArray);

console.log(i);

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

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

var printNumTwo;

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

if (i === 2) {

printNumTwo = function() {

return i;

};

}

}

console.log(printNumTwo());

Here the console will display the value 3.

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

let printNumTwo;

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

if (i === 2) {

printNumTwo = function() {

return i;

};

}

}

console.log(printNumTwo());

console.log(i);

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

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

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

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

function checkScope() {

var i = 'function scope';

if (true) {

i = 'block scope';

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

}

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

return i;

}

—---- resultado :

function checkScope() {

let i = 'function scope';

if (true) {

let i = 'block scope';

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

}

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

return i;

}

console.log(checkScope());

**Mutate an Array Declared with const**

If you are unfamiliar with const, check out [this challenge](https://www.freecodecamp.org/learn/javascript-algorithms-and-data-structures/basic-javascript/declare-a-read-only-variable-with-the-const-keyword).

The const declaration has many use cases in modern JavaScript.

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

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

const s = [5, 6, 7];

s = [1, 2, 3];

s[2] = 45;

console.log(s);

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

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

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

const s = [5, 7, 2];

function editInPlace() {

// Only change code below this line

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

// Only change code above this line

}

editInPlace();

—

const s = [5, 7, 2];

function editInPlace() {

// Only change code below this line

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

s.pop();

s.unshift(2);

// Only change code above this line

}

editInPlace();

console.log(s);

**Prevent Object Mutation**

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

Any attempt at changing the object will be rejected, with an error thrown if the script is running in strict mode.

let obj = {

name:"FreeCodeCamp",

review:"Awesome"

};

Object.freeze(obj);

obj.review = "bad";

obj.newProp = "Test";

console.log(obj);

The obj.review and obj.newProp assignments will result in errors, because our editor runs in strict mode by default, and the console will display the value { name: "FreeCodeCamp", review: "Awesome" }.

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

function freezeObj() {

const MATH\_CONSTANTS = {

PI: 3.14

};

// Only change code below this line

// Only change code above this line

try {

MATH\_CONSTANTS.PI = 99;

} catch(ex) {

console.log(ex);

}

return MATH\_CONSTANTS.PI;

}

const PI = freezeObj();

—---

function freezeObj() {

const MATH\_CONSTANTS = {

PI: 3.14

};

// Only change code below this line

Object.freeze(MATH\_CONSTANTS);

// Only change code above this line

try {

MATH\_CONSTANTS.PI = 99;

} catch(ex) {

console.log(ex);

console.log("Error");

}

return MATH\_CONSTANTS.PI;

}

const PI = freezeObj();

//console.log(freezeObj());

**Use Arrow Functions to Write Concise Anonymous Functions**

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

To achieve this, we often use the following syntax:

const myFunc = function() {

const myVar = "value";

return myVar;

}

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

const myFunc = () => {

const myVar = "value";

return myVar;

}

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

const myFunc = () => "value";

This code will still return the string value by default.

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

var magic = function() {

return new Date();

};

—-

var magic = function() {

return new Date();

};

console.log(magic());

const myFunc = (x) => x + 3 ;

console.log(myFunc(3));

—-

console:

Wed Apr 27 2022 16:19:28 GMT+0200 (hora de verano de Europa central)

6

—

const magic = () => new Date();

console.log(magic());

**Write Arrow Functions with Parameters**

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

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

doubler(4);

doubler(4) would return the value 8.

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

const doubler = item => item \* 2;

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

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

multiplier(4, 2);

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

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

var myConcat = function(arr1, arr2) {

return arr1.concat(arr2);

};

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

**¿Qué es Concat en JS?**

El método **concat**() se usa para unir dos o más arrays. Este método no cambia los arrays existentes, sino que devuelve un nuevo array

**Set Default Parameters for Your Functions**

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

Check out this code:

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

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

console.log(greeting());

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

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

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

// Only change code below this line

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

// Only change code above this line

—

// Only change code below this line

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

// Only change code above this line

console.log(increment(5,34));

**Use the Rest Parameter with Function Parameters**

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

Check out this code:

function howMany(...args) {

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

}

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

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

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

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

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

const sum = (x, y, z) => {

const args = [x, y, z];

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

}

si quieres **definir** una función cuyo número de parámetros de entrada puede ser variable, considera el uso de \***args** o \*\*kwargs

**map()**

El método map() crea un nuevo array con los resultados de la llamada a la función indicada aplicados a cada uno de sus elementos.

var numbers = [1, 5, 10, 15];

var doubles = numbers.map(function(x) {

return x \* 2;

});

// doubles is now [2, 10, 20, 30]

// numbers is still [1, 5, 10, 15]

**Math.sqrt(num)**

la raiz cuadrada del numero

**filter()**

El método filter() crea un nuevo array con todos los elementos que cumplan la condición implementada por la función dada.

const words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];

const result = words.filter(word => word.length > 6);

console.log(result);

// expected output: Array ["exuberant", "destruction", "present"]

**reduce()**

El método reduce() ejecuta una función reductora sobre cada elemento de un array, devolviendo como resultado un único valor.

const array1 = [1, 2, 3, 4];

// 0 + 1 + 2 + 3 + 4

const initialValue = 0;

const sumWithInitial = array1.reduce(

(previousValue, currentValue) => previousValue + currentValue,

initialValue

);

console.log(sumWithInitial);

// expected output: 10

RESULTADO

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

//const args = [x, y, z];

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

}

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

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

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

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

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

maximus would have a value of 89.

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

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

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

maximus would have a value of 89.

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

const spreaded = ...arr;

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