jQuery filter() method:

*var found\_ids = data.filter((i)=>{*

//*return i.id < 10;*

*return i.title === "qui est esse";*

*})*

jQuery grep() method:

*var found\_ids = $.grep(data, function(v) {*

*return v.title ==="qui est esse";*

*});*

jQuery foreach loop:

var result = [];

          data.forEach(element => {

*//console.log(element.id);*

            result.push(element.id + " ");

          });

jQuery Datepicker:

<head>

  <link

    rel="stylesheet"

    href="http://code.jquery.com/ui/1.12.1/themes/base/jquery-ui.css"

  />

  <script

    src="https://code.jquery.com/jquery-3.4.1.min.js"

    integrity="sha256-CSXorXvZcTkaix6Yvo6HppcZGetbYMGWSFlBw8HfCJo="

    crossorigin="anonymous"

  ></script>

  <script src="https://code.jquery.com/ui/1.12.1/jquery-ui.js"></script>

</head>

<body>

  <input type="text" id="date" />

  <div>

    <button id="btn">Click Me</button>

  </div>

  <div id="divID"></div>

</body>

<script>

  $(function() {

    $("#date").datepicker({

      minDate: 0,

      onSelect: function(date) {

        $("#date").datepicker("option", "minDate", date);

      },

      changeYear: true,

      changeMonth: true,

      dateFormat: "dd/mm/yy",

*//minDate: new Date("01/01/1900"),*

      maxDate: "+2Y"

    });

  });

7 ways to iterate over an Array:

<script>

  let arr = [5, 10, 15, 20, 25, 30];

  let i = 0;

*////////// While loop //////////////*

*//   while (i < arr.length) {*

*//     console.log(arr[i]);*

*//     i++;*

*//   }*

*////////// do While loop //////////////*

*//   do {*

*//     console.log(arr[i]);*

*//     i++;*

*//   } while (i < arr.length);*

*////////// for loop //////////////*

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

*//     console.log(arr[i]);*

*//   }*

*////////// for in loop //////////////*

*//   for (let i in arr) {*

*//     console.log(arr[i]);*

*//   }*

*////////// for of loop //////////////*

*//   for (let i of arr) {*

*//     console.log(i);*

*//   }*

*////////// foreach loop //////////////*

*//   arr.forEach(element => {*

*//     console.log(element);*

*//   });*

*////////// filter loop //////////////*

*//   var value\_found = arr.filter(ele => {*

*//     return ele > 20;*

*//   });*

*//   console.log(value\_found);*

*////////// Iterator //////////////*

*//   var itratr = arr[Symbol.iterator]();*

*//   console.log(itratr.next());*

</script>

Callback, Promises, Async/Await, Async/Await with Fetch:

Callback example:

const posts = [

  { title: "First post", body: "This is first post" },

  { title: "Second post", body: "This is first post" }

];

function getPosts() {

  setTimeout(() => {

    let output = "";

    posts.forEach(element => {

      output += `<li>${element.title}</li>`;

    });

    document.body.innerHTML = output;

  }, 1000);

}

function createPost(post, callback) {

  setTimeout(() => {

    posts.push(post);

    callback();

  }, 2000);

}

*//getPosts();*

createPost({ title: "Third post", body: "This is third post" }, getPosts);

Promise, Promise. All, Async/Await, Async/Await with Fetch example:

const posts = [

  { title: "First post", body: "This is first post" },

  { title: "Second post", body: "This is first post" }

];

function getPosts() {

  setTimeout(() => {

    let output = "";

    posts.forEach(element => {

      output += `<li>${element.title}</li>`;

    });

    document.body.innerHTML = output;

  }, 1000);

}

function createPost(post) {

*return* new Promise((resolve, reject) => {

    setTimeout(() => {

      posts.push(post);

      const error = false;

*if* (!error) {

        resolve();

      } *else* {

        reject("Error: Something went wrong");

      }

    }, 2000);

  });

}

*// createPost({ title: "Third post", body: "This is third post" })*

*//   .then(getPosts)*

*//   .catch(err => console.log(err));*

*///////////// Async / Await //////////////////////////////*

*// async function init() {*

*//   await createPost({ title: "Third post", body: "This is third post" });*

*//   getPosts();*

*// }*

*// init();*

*///////////// Async / Await with Fetch //////////////////////////////*

async function fetchUsers() {

  const res = *await* fetch("https://jsonplaceholder.typicode.com/users");

  const data = *await* res.json();

  console.log(data);

}

fetchUsers();

*///////////////////// Prmoise.all /////////////////////////////*

*// const promise1 = Promise.resolve("Hello World");*

*// const promise2 = 10;*

*// const promise3 = fetch("https://jsonplaceholder.typicode.com/users").then(res =>*

*//   res.json()*

*// );*

*// Promise.all([promise1, promise2, promise3]).then(value => console.log(value));*

**Bind (), Call () and apply () methods:**

# Bind();

**Syntax:**

function.bind(thisArg, optionalArguments)

For example:

Let’s suppose we have two person objects.

const john = {

  name: "John",

  age: 24

};

const jane = {

  name: "Jane",

  age: 22

};

Let’s add a greeting function:

function greeting() {

  console.log(`Hi i am ${this.name} and i am ${this.age} years old`);

}

We can use the bind method on the greeting function to bind the this keyword to john and jane objects.

For example:

const greetingJohn = greeting.bind(john);

*// Hi, I am John and I am 24 years old*

greetingJohn();

const greetingJane = greeting.bind(jane);

*// Hi, I am Jane and I am 22 years old*

greetingJane();

Here greeting.bind(john) creates a new function with this set to john object, which we then assign to greetingJohn variable. Similarly for greetingJane.

## Bind() can also accept arguments

We can also pass extra arguments to the bind method. The general syntax for this is

function.bind(this, arg1, arg2, ...).

For example:

function greeting(lang) {

  console.log(`${lang}: I am ${this.name}`);

}

const greetingJohn = greeting.bind(john, "en");

greetingJohn();

const greetingJane = greeting.bind(jane, "es");

greetingJane();

# Call()

The call method sets the this inside the function and immediately executes that function.

The difference between call() and bind() is that the call() sets the this keyword and executes the function immediately and it does not create a new copy of the function, while the bind() creates a copy of that function and sets the this keyword.

## Syntax:

function.call(thisArg, arg1, agr2, ...)

For example:

const john = {

  name: "John",

  age: 24

};

const jane = {

  name: "Jane",

  age: 22

};

function greeting() {

  console.log(`Hi, I am ${this.name} and I am ${this.age} years old`);

}

*// Hi, I am John and I am 24 years old*

greeting.call(john);

*// Hi, I am Jane and I am 22 years old*

greeting.call(jane);

Above example is similar to the bind() example except that call() does not create a new function. We are directly setting the this keyword using call().

## Call () can also accept arguments

Call() also accepts a comma-separated list of arguments. The general syntax for this is

 function.call(this, arg1, arg2, ...)

For example:

function greet(greeting) {

  console.log(`${greeting}, I am ${this.name} and I am ${this.age} years old`);

}

const john = {

  name: "John",

  age: 24

};

const jane = {

  name: "Jane",

  age: 22

};

*// Hi, I am John and I am 24 years old*

greet.call(john, "Hi");

*// Hi, I am Jane and I am 22 years old*

greet.call(jane, "Hello");

# Apply ()

The apply () method is similar to call (). The difference is that the apply () method accepts an array of arguments instead of comma separated values.

## Syntax:

function.apply(thisArg, [argumentsArr])

For example:

function greet(greeting, lang) {

  console.log(lang);

  console.log(`${greeting}, I am ${this.name} and I am ${this.age} years old`);

}

const john = {

  name: "John",

  age: 24

};

const jane = {

  name: "Jane",

  age: 22

};

greet.apply(john, ["Hi", "en"]);

*//en*

*// Hi, I am John and I am 24 years old*

greet.apply(jane, ["Hola", "es"]);

//es

//Hola, I am Jane and I am 22 years old

# Conclusion

We have learned that how this keyword behaves differently in JavaScript than in other object-oriented languages. The call, bind and apply methods can be used to set the this keyword independent of how a function is called.

The bind method creates a copy of the function and sets the this keyword, while the call and apply methods sets the this keyword and calls the function immediately.

# Understand var, let and const keywords in JavaScript

## THE VAR KEYWORD

Function scoping is best practice to use the **var** keyword because when you define a variable inside function’s scope using the **var** keyword, this variable means it’s the local variable that outside cannot access. Following this code:

function foo(){

var x = "inside";

console.log(x);

}

foo(); //inside

console.log(x) // Error, x is not defined

As you can see, we tried to log out the value of x but because x is treated as a local variable with **var** keyword thus the error will appear unsurprisingly.

With the **var** keyword, we totally can re-assign the value of variables:var x = 5;

x = 6;

console.log(x); // 6

function greeting() {

var a = "Hello";

a = "Hello JavaScript!";

console.log(a);

}

greeting();

Not only were able to re-assign variables by using the **var** keyword, but it can also be re-declared even when you use strict mode:

'use strict';

var x = 30;

var x = 70;

console.log(x); // 70

var y = "Java";

var y = "JavaScript";

console.log(y); // JavaScript

When using the **var** keyword, you always need to remember put it in scope which means this variable is a local variable, but if in case you put it outside of a function or scope it would become a global variable that can be accessed anywhere in your program:

var x = "outside";

function inside(){

console.log(x);

}

inside(); // outside

/\* Because declare variable outside of function scope

this variable will become global and it can be

re-assigned inside the function scope and when we log

the value of y, it has been changed, consider the code

below:

\*/

var y = 30;

function number(){

y = 20;

var z = 50;

console.log(y + z);

}

number(); // 70

console.log(y); // 20

## THE LET KEYWORD

The let keyword is similar to var. The **let** keyword is used to declare a local variable inside a block (like **var**), if you declare variable used let keyword outside of a block, it’s also being a global variable (also, like var) and if you want to re-assign the value of your variable which defined with let keyword, you totally can do it (as well **var**), consider the following codes below:

function name(){

let x = "My name is Jessica"; // Local variable

console.log(x);

}

name(); // My name is Jessica

let y = 27; // Global variable

function sumUp(){

let z = 23;

console.log(y+z);

}

sumUp(); // 50;

let q = 13;

function divide(){

let w = 2;

q = 14; // Re-assign

console.log(q/w)

}

divide(); // 7

Nonetheless, the difference between **var** and **let** are, with **var** you **can re-declare** variable, but you **can’t** do it with **let**. A variable with the **var** keyword is used in a loop is visible in a **whole function**, with **let** keyword it’s just **visible in loop scope**. Consider the code below:

let x = 20;

let x = 30; // Uncaught SyntaxError: Identifier 'x' has already been declared

function aLoop(){

for(let i = 0; i < 20; i++){

console.log(i);

}

// i is not visible here.

}

## THE CONST KEYWORD

In general, both var and let are used to declare variables that can be re-assigned, but sometimes you don’t want to re-assign values to your variable. It’s time when the **const** keyword comes in handy, when you declare a variable with the **const** keyword, meaning this variable might be a **constant** and its value cannot be changed once initialized.

onst x = 2;

const x = 3; // Error, cannot be re-declare

const z = 5;

z = 7; // Error, cannot be re-assign

const a = [2, 3, 4, 5];

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

console.log(a[i]+ 1);

} // Output: 3, 4, 5, 6

/\* Variable declared with const keyword

cannot be re-assigned, but it's still

mutable.

\*/

If you declare a variable using const keyword outside of a block context, it’s also become global:

const x = 69;

function count(){

console.log(x + 96);

}

count(); // 165

### CONCLUSION

With JavaScript ES6, there are a few different ways that help you declare a variable, depending on your purpose to choose the right keyword, remember **var** keyword is weakest, **let** is the popular keyword that is utilized for, maybe re-assigned variable but cannot re-declared. The **const** keyword is the best if you want to declare a constant or something that you firmly want that its values cannot be assigned.

|  |  |  |  |
| --- | --- | --- | --- |
| Variable Type | Can Re-assign the values | Can Re-declare the variable | Can act as globally |
|  |  |  |  |
| Var | Yes | Yes | When using the **var** keyword, you always need to remember put it in scope which means this variable is a local variable, but if in case you put it outside of a function or scope it would become a global variable that can be accessed anywhere in your program |
|  |  |  |  |
|  |  |  |  |
| Let | Yes | No | Yes, when declared outside the function block. |
| Const | No | No | If you declare a variable using const keyword outside of a block context, it’s also become global: |

# Top 10 Handy JavaScript Array Methods You Should Know

## 1. MAP()

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

const array2 = array.map(x => x \* 2);

console.log(array2); // [ 2, 4, 6, 8 ]

const array3 = array.map(x => x \* x);

console.log(array3); // [ 1, 4, 9, 16 ]

## 2. FILTER()

The filter() method creates a new array with elements that pass the test implemented by the provided callback function.

const array = [5, 6, 7, 8, 9, 10];

const filteredArray = array.filter(number => number > 7);

console.log(filteredArray); // [ 8, 9, 10 ]

const anotherArray = [49, 21, 83, 62, 51, 54, 69];

const multiplesOfSeven = anotherArray.filter(number => number % 7 == 0);

console.log(multiplesOfSeven); // [ 49, 21 ]

## 3. SORT()

The sort() method is a built-in JavaScript method that allows you to sort elements of the array in place and returns the sorted array. The default sorting order is ascending, it first converts elements into strings, then comparing their sequences of UTF-16 code values.

const letters = ["X", "Z", "A", "C", "D", "F", "J", "N", "H"];

letters.sort();

console.log(letters); // [ 'A', 'C', 'D', 'F', 'H', 'J', 'N', 'X', 'Z' ]

If we want to sort elements of numbers in ascending order:

const numbers = [6000, 2000, 1000000, 20, 1999];

numbers.sort((a, b) => a - b);

console.log(numbers); // [ 20, 1999, 2000, 6000, 1000000 ]

Sorting elements in descending order:

const numbers = [6000, 2000, 1000000, 20, 1999];

numbers.sort((a, b) => b - a);

console.log(numbers) // [ 1000000, 6000, 2000, 1999, 20 ]

## 4. REDUCE()

This method takes a function which has an accumulator and a value as an argument and the reduce() method applies this function to every element of the array from left to right, resulting in a single output value.

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

const b = a.reduce((accumulator, currentValue) => accumulator + currentValue);

console.log(b); // 1 + 2 + 3 + 4 + 5 = 15

You also can provide an optional initial value passed to the function:

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

const reducedNumber = numbers.reduce((accumulator, currentValue) => accumulator \* currentValue, 5);

console.log(reducedNumber); // 5 \* 1 \* 2 \* 3 \* 4 \* 5 = 600

## 5. FOREACH()

The forEach() method calls a function once for each element in an array, in order and it always returns undefined value at the end.

const languages = ["JS", "Python", "Java", "Haskell"];

const array = [];

languages.forEach(x => array.push(x + " is awesome!"));

console.log(array);

/\*

[ 'JS is awesome!',

'Python is awesome!',

'Java is awesome!',

'Haskell is awesome!' ]

\*/

const people = [

{ name: "Nam", age: 69 },

{ name: "Justin", age: 23 },

{ name: "Austin", age: 28 }

]

people.forEach(person => console.log(person.name));

/\*

Nam

Justin

Austin

\*/

## 6. REVERSE()

This method reverses the order of the array in-place, the first element will become the last, and vice versa.

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

const reversed = numbers.reverse();

console.log(reversed); // [ 50, 40, 30, 20, 10 ]

const letters = ["A", "B", "C", "D", "E"];

const reversedOrder = letters.reverse();

console.log(reversedOrder); // [ 'E', 'D', 'C', 'B', 'A' ]

## 7. INDEXOF()

The indexOf() method returns the first index at which a given element can be found in the array, or -1 if it is not present.

const names = ["Nam", "Justin", "Juan", "Christopher", "Michael", "Nam"];

console.log("Index of 'Nam' is " + names.indexOf('Nam'));

// 0

You also can specify the second parameter which is the start index, where the method begins to find the index’s element, for example, we want to find the index of the element 'Nam' from the second index till the end:

const names = ["Nam", "Justin", "Juan", "Christopher", "Michael", "Nam"];

console.log("Index of 'Nam' is " + names.indexOf('Nam', 1)); // Index of 'Nam' is 5

This method will return -1 in case it cannot find the index of the element that you are looking for:

const nums = [1, 2, 3, 4, 5, 6, 7, 8];

console.log(nums.indexOf(9)); // -1

## 8. CONCAT()

The concat() method is used to merge two or more arrays. But it doesn’t change the existing arrays, instead, it returns a new array after merging.

const arr1 = ["Where", "the", "north", "wind"];

const arr2 = ["meets", "the", "sea"];

const arr3 = arr1.concat(arr2);

console.log(arr3); // [ 'Where', 'the', 'north', 'wind', 'meets', 'the', 'sea' ]

const nums = [1, 2, 3];

const letters = ['a', 'b', 'c'];

console.log(nums.concat(letters)); // [ 1, 2, 3, 'a', 'b', 'c' ]

console.log(nums); // [ 1, 2, 3 ]

console.log(letters); // [ 'a', 'b', 'c' ]

## 9. JOIN()

The join() method creates and returns a new string which is the concatenation of all elements in the array it applies, it accepts one parameter as a separator to separate elements in this array.

const a = ["Where", "the", "north", "wind", "meets", "the", "sea"];

// use "" as the separator for array elements

console.log(a.join("")); // Wherethenorthwindmeetsthesea

// use " " as the separator for array elements

console.log(a.join(" ")); // Where the north wind meets the sea

// use "-" as the separator for array elements

console.log(a.join("-")); // Where-the-north-wind-meets-the-sea

If the array has only one item, then that item will be returned without using the separator.

const x = ["Something"];

console.log(x.join()); // Something

## 10. INCLUDES()

The includes() method returns true if the array contains a certain element, false if not.

const arr1 = [1, 2, 3, 4, 5];

console.log(arr1.includes(1)); // true

const arr2 = ['one', 'two', 'four'];

console.log(arr2.includes('three')); // false

This method accepts the second parameter, which is the position in this array at which to begin searching for the value we want to find.

const arr3 = [1, 2, 3];

// find value 1 in the arr3 starting from index 2

console.log(arr3.includes(1, 2)); // false