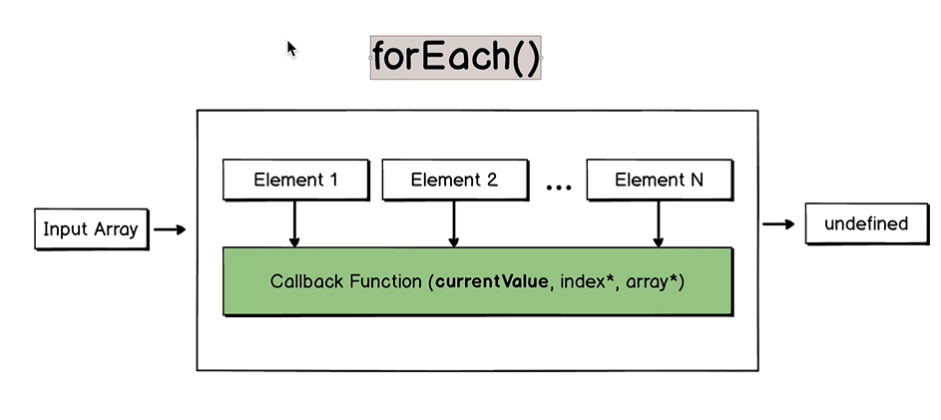
**Section 15**: Array Helper Methods

forEach()



Takes in an input array and return value is “undefined”, that is it does not return anything.

Takes each element one by one and each element is passed to the callback function as ‘currentValue’ variable.

Main purpose of forEach() method is to perform some action on each element of the array.

Arrow functions are mostly used as callback functions for other functions.

const myArray = [1, true, "abc"]

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

console.log(myArray[i]);

}

myArray.forEach( element => console.log(element) )

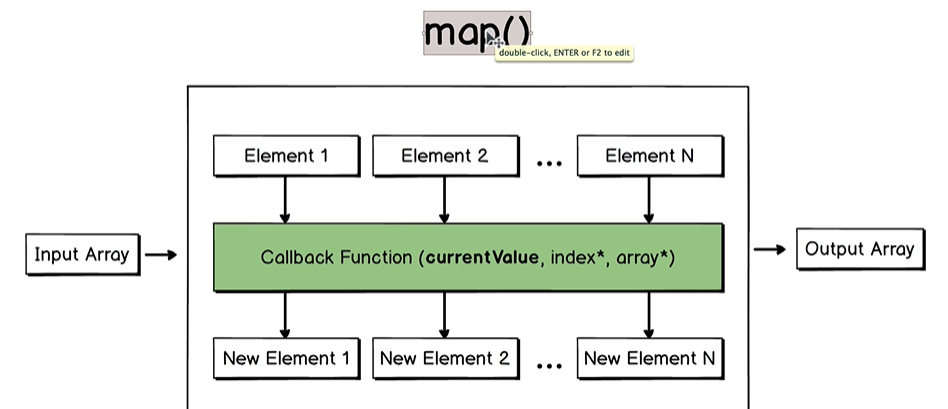
const myCities = [ “abc”, “def”, “xyz” ]

myCities.forEach( (element, index) => console.log(arrayInfo(element, index) ) );

In forEach() method, first argument is the array element and second is the index number of that element.

**Lecture 158**:

map() function



Each element is passed to the callback function. The structure of the callback function is the same as is for forEach() method.

Here it matters what the callback function returns for every element.

Input array is not mutated.

map() method creates a brand new array.

**Lecture 159**:

const myNumbers = [1, 5, 7];

let mySquareNumbers = [];

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

mySquareNumbers.push(myNumbers[i] \* myNumbers[i] ) ;

}

const myNumbers = [1, 5, 7];

let mySquareNumbers = myNumbers.map(element => element \* element);

const myNumbers = [1, 5, 7];

const squareNumbers = element => element \* element;

let mySquareNumbers = myNumbers.map(squareNumbers);

// There is implicit return of “element \* element” above.

const myNumbers = [1, 5, 7];

const squareNumbers = (element, index) => {

console.log("Element at the index " + index + " is " + element);

return element \* element; // explicit return

}

let mySquareNumbers = myNumbers.map(squareNumbers);

// mySquareNumbers references to an array.

Interesting:

const myNumbers = [1, 5, 7];

let mySquareNumbers = myNumbers.map(Math.pow);

This will return the array with elements, [ 1^0, 5^1, 7^2 ], that is [1, 5, 49]

And so on.

These values of 0, 1, 2, … are basically indices which are used as powers.

pow() takes 2 parameters, number and power. Index number gets passed as the power here.

(base value, exponent value)

Be careful when we pass any function by its name to the map() function.

In each iteration, map() method will pass to the callback function 3 arguments that are element, index and initial array.

“index” in Math.pow() function will be treated as exponent value.

map() function requires one argument which is the callback function.

And the callback function for map() could be called with maximum of 3 arguments, element, index and initial array.

**Lecture 161**:

Convert array of JSON objects to array of Javascript objects.

const postsJSON = [

'{"postsId": 1355, "commentsQuantity": 5}',

'{"postsId": 5131, "commentsQuantity": 13}',

'{"postsId": 6134, "commentsQuantity": 2}',

];

/\* From the "postsJSON" array create "posts" array that will consist

of Javascript objects.

\*/

var posts = [];

// With for-loop

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

posts.push(JSON.parse( postsJSON[i] ));

}

console.log(posts);

// Using map()

const posts = postsJSON.map(JSON.parse);

console.log(posts);

Using ‘const’, we can mutate the ‘posts’ array.

// Passing arrow function to map() function,

const posts = postsJSON.map(post => JSON.parse(post));

console.log(posts);

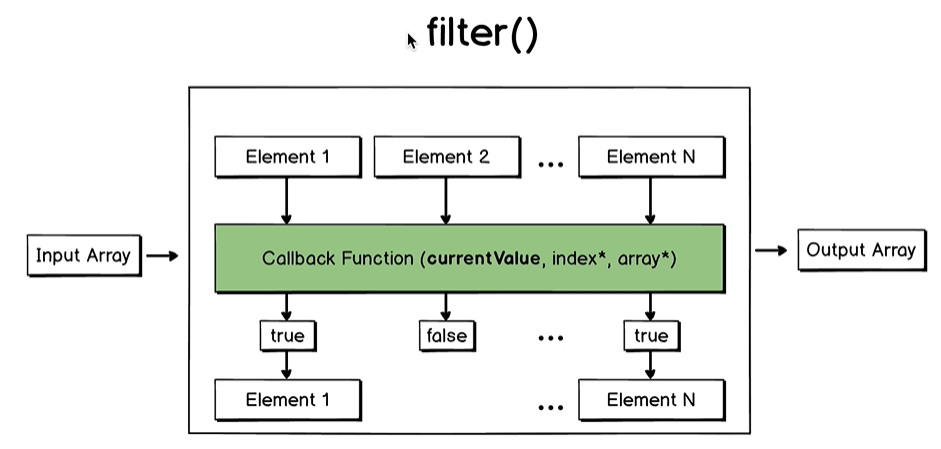
// Here we get correct even if we pass just the JSON.parse function to the map() function as compared to when we just passed Math.pow function before.

With JSON.parse, we do not have the problem that we had with Math.pow.

JSON.parse expects function as the second argument but because we pass a number which is the index of the element, this argument is simply ignored by the parse() method.

**Lecture 162**:

filter()



Elements for which the callback function returns true is pushed to the output array.

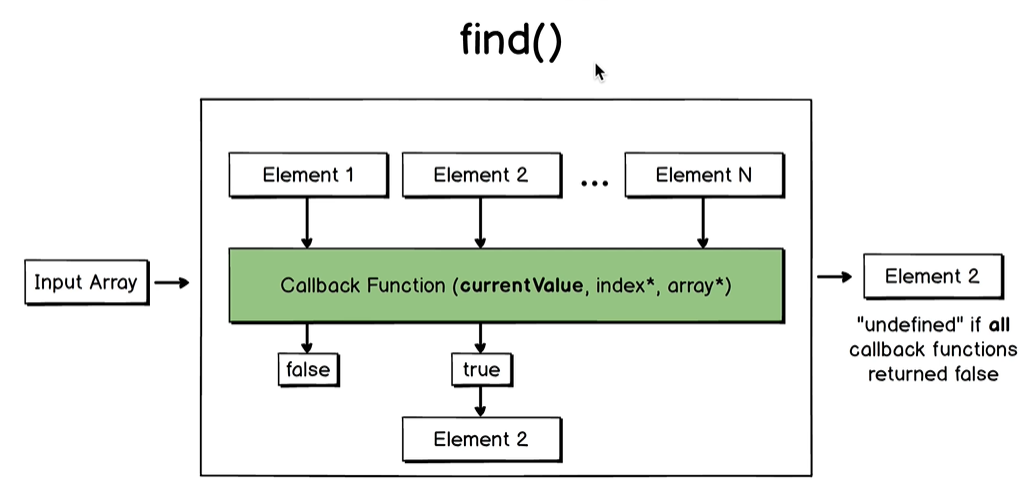
Callback function does filtering of the input elements in the input array.

Elements are not changed. Input array is not mutated.

const myNumbers = [ 10, 25, 56, 100, 5];

const filteredNumbers = myNumbers.filter(num => num > 10)

**Lecture 164**: find()



find() returns an element or undefined.

find() method returns the first element where the callback function returns true.

const myArray = [10, [], {}, "abc", true, 15 ] ;

const result = myArray.find(element => type of element === "number")

// returns 10

**Lecture 166**:

const posts = [

{ postId: 1355, commentsQuantity: 5},

{ postId: 5131, commentsQuantity: 13},

{ postId: 6134, commentsQuantity: 2}

] ;

/\* Create a function "findSinglePost" that will have 2 parameters - "postId" and

"posts" and will return object with matched "postId".

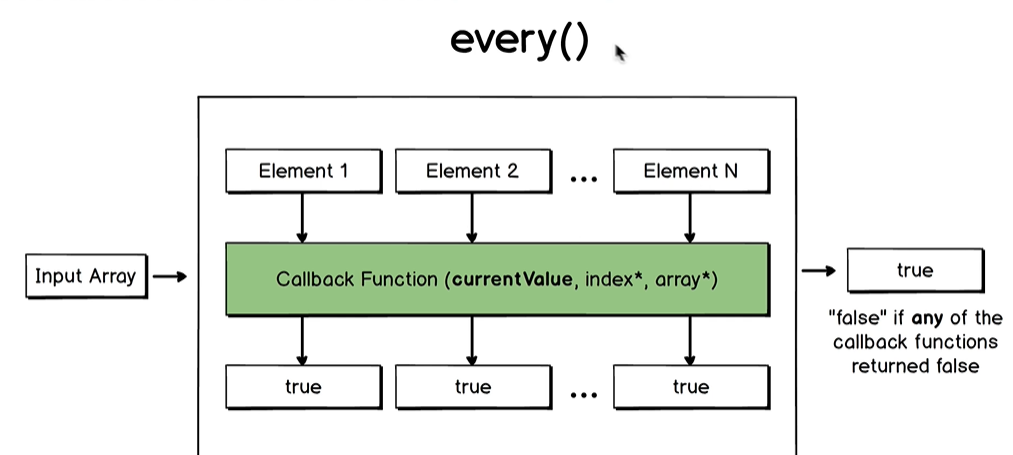
If post was not found - return "undefined".

\*/

const findSinglePost = (postId, posts) => posts.find(post => post.postId === postId) ;

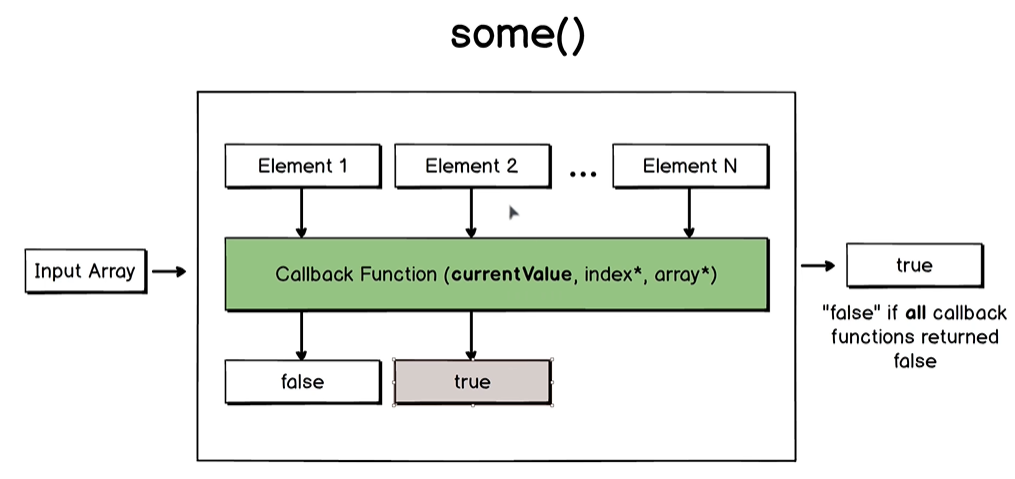
console.log(findSinglePost(6134, posts));

**Lecture 168**: every() and some(), boolean result.



When we want to check whether all elements of the array have same properties or characteristics.

some()



When we want to check whether atleast one element matches a specific criteria.

const myNumbers = [ 3, -5, 1, 10, -7 ] ;

const isPositiveNumber = element => typeof element === "number" && element > 0;

const allPositivesCheck = myNumbers.every(isPositiveNumber) // false

const somePositivesCheck = myNumbers.some(isPositiveNumber) // true

const items = [

{ title: "Computer", quantity: 10 },

{ title: "Phone", quantity: 3 },

{ title: "Headphones", quantity: 15 }

];

// Check whether all items have quantity > 5

if(items.every(item => item.quantity > 5 )){

console.log("All items are available");

}

if (items.every(item => item.quantity > 0 ) && items.some(item => item.quantity <= 5 )){

console.log("Some items may be sold soon");

}

**Lecture 171**:

const a = [5, "abc", 10, 1];

const b = [4, 10, 14, 25, 25, 50];

const c = [150, 132, 80, 40];

const d = [15, 26, 10, 23, 85];

/\*

Create a function "arrayCheck" with one parameter - "inputArray".

If atleast one element in the array is not a number - return "Some elements are not numbers"

If numbers in the array are sorted in ascending order -

return "Array is sorted in ascending order"

If numbers in the array are sorted in descending order -

return "Array is sorted in descending order"

If array is not sorted - return "Array is not sorted"

\*/

const arrayCheck = inputArray => {

if(inputArray.some(element => type of element !== "number")) {

return "Some elements are not numbers"

}

// First we need to check whether all the elements are numbers.

if(inputArray.every(element, index, array) => index > 0 ? element >= array[index-1]: true ) {

return "Array is sorted in ascending order";

} ;

if(inputArray.every(element, index, array) => index > 0 ? element <= array[index-1]: true ) {

return "Array is sorted in descending order";

} ;

return "Array is not sorted" ;

}

**Lecture 173:**

Are Arrays equal or not?

All variable types in Javascript are split into 2 categories: Primitives and reference types.

Arrays and objects are reference types in javascript.

Variable holds a pointer in the memory where the value is stored.

const a = [1, 2, 3];

const b = [1, 2, 3];

console.log(a==b) ; // false

Here pointers are compared and not values.

const arraysAreEqual = (arrayOne, arrayTwo) =>

arrayOne.length === arrayTwo.length &&

arrayOne.every((element, index) => element === arrayTwo[index] )

**Lecture 175**:

=== (3 equals) checks the value as well as the type.

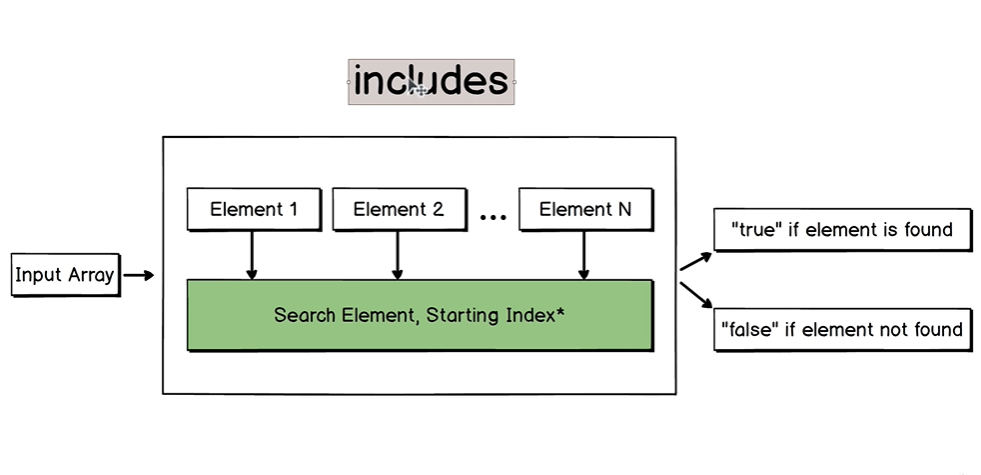
Check if an element exists in the array.

const elementFound = ( inputArray, searchElement) =>

inputArray.some( element => element === searchElement)

some() and includes() methods find elements that are of primitive types such as strings, numbers, booleans, undefined or null.

**Lecture 176**:



includes() method does not require a callback function.

Start looking from the specified index (optional).

includes() works perfectly with primitive types.

Returns false for reference types.

**Lecture 179**: includes() with Array or Object

If we pass an object or array to the includes() method, then convert each element in the array to the string and then apply “includes” method with argument that will be also converted to the string.

(Argument will be an element of the array).

A JSON object is a string.

JSON.stringify() is opposite of JSON.parse() and converts any object or array to string.

Check the type using type of operator.

typeof {} “object”

typeof [] “object”

There is only one reference type in Javascript, Object.

const elementIsIncluded = (searchElement, array) => {

if(typeof searchElement !== "object"){

return array.includes(searchElement);

}

if(typeof searchElement === "object"){

return array.map(element => JSON.stringify(element)).includes

(JSON.stringify(searchElement))

}

};

{title: “Apple”, quantity: 25}

JSON.stringify returns

**“**{**“**title**”**: “Apple”, **“**quantity**”**: 25}**”**

Double quotes are added around each key in each key-value pair.

[“flexbox”, “css”]

JSON.stringify for this returns

**“**[“flexbox”, “css”]**”**

**Lecture 181:**

Push element to the array if the element does not exist.

Assume that input array contains only primitive variable types.

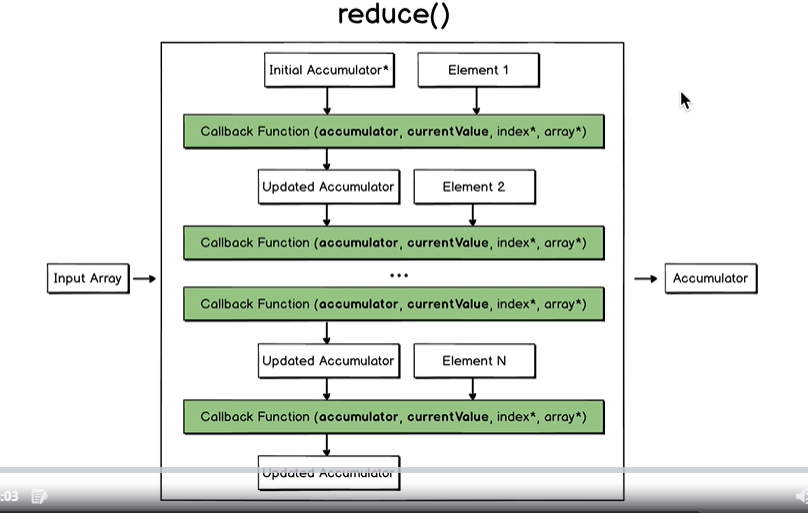
Will use includes()

const pushIfUnique = (inputArray, newElement) =>

inputArray.includes(newElement) ? console.log(newElement + " already exists") : inputArray.push(newElement)

For array or objects, we can use JSON.stringify() function.

**Lecture 182**: reduce()



reduce() method maintains state called as accumulator while looping through the elements in the array.

It takes an input array and reduces the array to one value.

This value may have any variable type in Javascript, for example it can be “number”, “string”, “boolean” or “object”

This reduce() method has 2 parameters, callback function like map(), forEach(), etc and initial accumulator.

2nd parameter is optional. If initial accumulator is present in the reduce() method call, then iteration will start from first element in the array.

And initial accumulator and first element will be passed to the callback function.

The callback function has 2 mandatory parameters, accumulator and currentValue and 2 optional parameters ‘index’ and ‘array’.

In this method, ‘accumulator’ is the new parameter.

The callback function performs some actions with accumulator and currentValue.

Its returns a result called as ‘Updated Accumulator’.

This updated accumulator and element 2 are passed again to the callback function.

When we reach the last element of the array, then this final element and latest updated accumulator are passed to the callback function to return a final updated accumulator as the result of reduce() helper method.

If initial accumulator is absent in the reduce() method call, then the iteration will be started from element 2.

In the 2nd iteration, updated accumulator will be element 1.

We can reduce array of objects to a single object with several properties.

We can also reduce array of objects to array of elements, for example to array of numbers or array of strings.

We pass the accumulator and number to the callback function and callback function is the first parameter of reduce().

The second argument is the initial accumulator.

To find the sum of all numbers in an array,

const myNumbers = [5, 10, 3, 5];

const sum = arrayOfNumbers =>

arrayOfNumbers.reduce( (accumulator, number) => accumulator + number, 0)

console.log(sum(myNumbers))

If we are using { } for the arrow function body, then we have to explicitly use the “return” keyword.

For the first iteration, accumulator is 0.

If we omit the second argument in the reduce() function, then initial accumulator is the first element.

arrayOfNumbers.reduce( (accumulator, number) => accumulator + number)

And the iteration starts from the 2nd element of the array.

// Convert an array of objects to an array of single properties.

const persons = [

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

{ name: "J", age: 21 },

{ name: "K", age: 24 }

];

const personNames = arrayOfPersons =>

arrayOfPersons.reduce((names, person) => {

names.push(person.name)

return names

}, [ ]);

// names is the accumulator. And it is passed to the next iteration of the reduce() method.

// Remove duplicates.

const fruits = [ "banana", "orange", "apple", "apple", "orange", "mango" ]

const uniqueFruits = arrayOfFruits =>

arrayOfFruits.reduce( (uniqueElements, fruit) => {

if(!uniqueElements.includes(fruit)){

uniqueElements.push(fruit)

return uniqueElements

}

}, [ ])

console.log(uniqueFruits(fruits))

In the last 2 examples, it was necessary to pass in the initial accumulator for correct behaviour.

**Lecture 185: Reduce Array of Objects**

const popularPostIds = (posts, minimalCommentsQty) => {

posts.reduce( (postIds, post) => {

if(post.comments >= minimalCommentsQty){

postIds.push(post.postId)

return postIds // return so as to use in the next iteration

}

}, [ ]) ;

}

Reduce Array of Objects **to an Object**:

const products = [

{ title: "Phone Case", price: 23, quantity: 2, category: "Accessories" },

{ title: "Android Phone", price: 150, quantity: 1, category: "Phones" },

{ title: "Headphones", price: 78, quantity: 1, category: "Accessories" },

{ title: "Sports Watch", price: 55, quantity: 2, category: "Watches" }

]

const quantityByCategories = products => {

products.reduce( (counts, product) => {

counts[product.category] =

(counts[product.category] || 0) + product.quantity;

return counts

} , [ ])

}

If the ‘category’ key is absent in the counts array, then new key is created and its value is initialized to 0 + product.quantity.

We need parentheses because of difference in priorities of + and || operators.

Priority of + operator is higher than priority of || operator.

If printing an object gives Object, then convert it into a string using JSON.stringify() function.

**Lecture 188**: sort()

So far, we have covered the following helper methods:

forEach

map

filter

find

every

some

includes

reduce

const myNumbers = [10, 5, 7, 12, 20]

console.log(myNumbers.sort());

// returns [10, 12, 20, 5, 7]

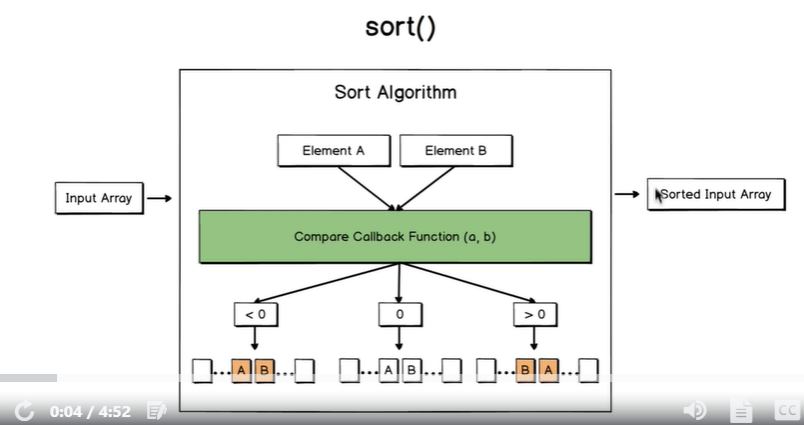
For correct usage, pass a callback function to sort() function.

console.log(myNumbers.sort( (a, b) => a - b ) );

Insertion sort is used to implement sort() method in most browsers.

Bubble and insertion sort: Stable

**Lecture 191**:



sort() method mutates original array and does not create brand new array.

Ecmascript specification does not explicitly say about a particular sorting algorithm.

If callback function returns 0, then order of elements won’t be changed.

**Lecture 192**: sort() withour arguments

There is default callback function which runs. Converts each element to a string inside of the sort() method.

“5” > “10” in string representation.

**Lecture 193**:

Different javascript engines use different sorting algorithms.

To sort array in descending order, swap ‘a’ and ‘b’ operands.

myNumbers.sort( (a, b) => b - a )

(a, b) represent each pair of elements passed to the callback function.

In Google chrome, if quantity of elements <=10, insertion sort is used.

**Lecture 194:**

Mozilla: Insertion sort (spider monkey js engine)

Safari: Merge sort

Node.js: Insertion sort (Qty<=10)

Node.js (node filename.js) and Google chrome use same V8 javascript engine, here the sort is stable.

**Lecture 196:**

sort() Array of objects

**Lecture 198**:

Array having more than 10 elements.

const sortPersonsByFriendsQty = persons => {

let it = 0;

persons.sort((a, b) => {

it++;

return a.friendsQty - b.friendsQty;

});

console.log("Quantity of iterations is : " + it)

return persons

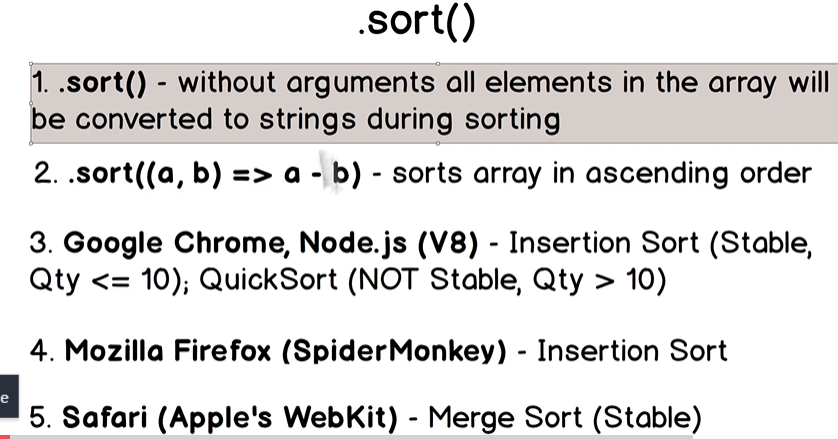
}

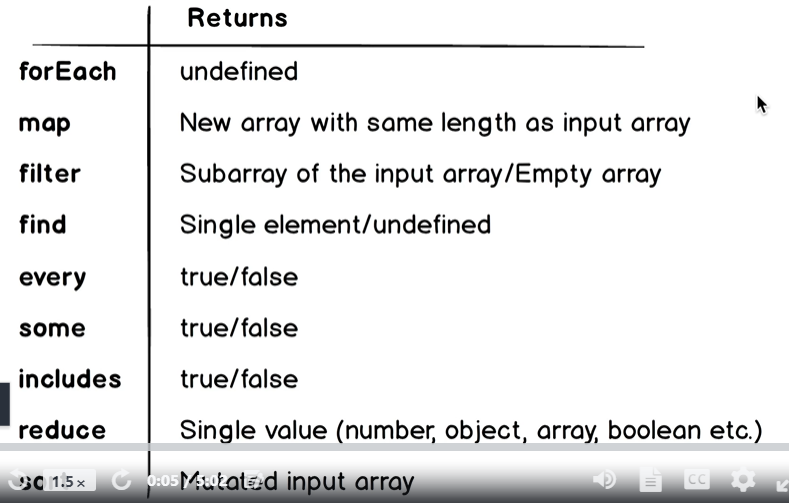
// Pass By Reference

**Google Chrome, Node.js**: Quick Sort (Qty >10), not stable (V8 JS engine)

Safari: Merge Sort, stable

Firefox (SpiderMonkey js engine): Insertion Sort, stable





Higher order functions. Methods of the array prototype. Available for each array.

Never use traditional for loop.

Always use array helper methods.