Rajesh Pillai Follow
Founder and Director of Algorisys Technologies. Passionate about nodejs, deliberate practice, .net, databases and all things javascript.
Apr 10 · 13 min read

All about JavaScript Arrays in 1 article

Everything you ever needed to know about JavaScript Arrays and my favorite part is the **reduce() method**.

No more feeling upside down when working with 'arrays'

After writing my article on All about JavaScript Functions in 1 article, my next objective is to cover the arrays in depth on similar lines.

This story is part of a larger open source curriculum

Mastering front end engineering in 12 to 20 weeks for beginners and experienced alike.

Arrays are a neat way to store continuous items in memory in a single variable. You can access elements in the array by its index. The index of the array starts with 0.

Let us create an empty array in multiple ways.

```
let tasks = [];
let projects = new Array();
```

Arrays in JavaScript are denoted by square bracket []. To get the length of array or count of elements in the array we can use the length property.

```
tasks.length;   // returns 0;
projects.length; // return 0;
```

One thing that you can do with an Array constructor that you cannot do with an array literal created with bracket notation is that you can set an initial length.

For example.

```
let projects = new Array(10);


console.log(projects.length); // Outputs 10
```

Recommended way is to use [] notation to create arrays.

We will talk on some performance issues, and creating holes in array that result in lower performance etc, in later part of the article as an update.

So, all exercises going forward will use the [] notation.

Let us store some values in array and access it by index

NOTE: You can store anything in an array, but for best performance use only one type of data in one array, for eg. array in text, array of numbers, array of objects etc. Don't mix types (atleast try to avoid mixing types)

```
let projects = ['Learn Spanish', 'Learn Go', 'Learn
Erlang'];
```

We can access the array elements by index as shown below.

```
console.log(projects[0]);  // Outputs  'Learn Spanish'
console.log(projects[2]);  // Outputs  'Learn Erlang';
console.log(projects[3]);  // Outputs  'undefined'
```

Adding items to array

Adding items to the end of an array (push method)

Let us add some items to the end of an array. We use the push() method to do so.

```
projects.push("Learn Malayalam");

console.log(projects);

// ["Learn Spanish", "Learn Go", "Learn Erlang", "Learn
Malayalam"]
```

NOTE: Push() method mutates the array. Also, check Pop() method to remove element from array.

Adding items to the beginning of an array (unshift)

Let us add some items to the end of an array. We use the unshift() method to do so.

```
projects.unshift("Learn Tamil");
console.log(projects);

// ["Learn Tamil", "Learn Spanish", "Learn Go", "Learn
Erlang", "Learn Malayalam"]
```

NOTE: unshift() mutates the array.

To add multiple items to the beginning of the array, just pass the required arguments to unshift method.

```
projects.unshift("Learn French", "Learn Marathi");


console.log (projects);


// ["Learn French", "Learn Marathi", "Learn Tamil", "Learn
Spanish", "Learn Go", "Learn Erlang", "Learn Malayalam"]
```

Adding items to the beginning of an array ES6 (spread operator)

```
let projects = ['Learn Spanish', 'Learn Go', 'Learn
Erlang'];


projects = ['Learn Malayalam', ...projects];


console.log(projects);
//Outputs-> ['Learn Malayalam', 'Learn Spanish', 'Learn Go',
               'Learn Erlang']
```

Adding items to the end of an array ES6 (spread operator)

```
let projects = ['Learn Spanish', 'Learn Go', 'Learn
Erlang'];


projects = [...projects,'Learn Malayalam'];


console.log(projects);
//Outputs-> ['Learn Spanish', 'Learn Go',
               'Learn Erlang','Learn Malayalam']
```

NOTE: The es6 approach does not mutate the array and gives a new updated back.

Remove first item from the array -> shift() method

The shift() method removes the first element from an array and returns that removed element. This method changes the length of the array. It returns undefined if the array is empty.

```
let numbers = [1,2,3,4];
let firstNo = numbers.shift();
console.log(numbers); // [2,3,4];
console.log(firstNo); // 1
```

Remove portion of an array, slicing -> slice() method

Slice method MDN Reference is pretty useful method as it enables to cut the array from any position.

The **`slice()`** method returns a shallow copy of a portion of an array into a new array object selected from `begin` to `end` (`end` not included). The original array will not be modified.

```
arr.slice([begin[, end]])
```

Let us have a look at some examples.

```
var elements = ['Task 1', 'Task 2', 'Task 3', 'Task 4',
'Task 5'];

elements.slice(2) //["Task 3", "Task 4", "Task 5"]


elements.slice(2,4)  // ["Task 3", "Task 4"]


elements.slice(1,5) // ["Task 2", "Task 3", "Task 4", "Task
5"]
```

Remove /adding portion of an array -> splice() method

The `splice()` method changes the contents of an array by removing existing elements and/or adding new elements. Be careful, splice() method mutates the array.

Detailed reference here at MDN splice method.

```
array.splice(start[, deleteCount[, item1[, item2[, ...]]]])
```

Parameters

`start` Index at which to start changing the array (with origin 0).

`deleteCount` (Optional)An integer indicating the number of old array elements to remove.

`item1, item2, ...` (Optional)The elements to add to the array, beginning at the `start` index. If you don't specify any elements, `splice()` will only remove elements from the array.

Return value

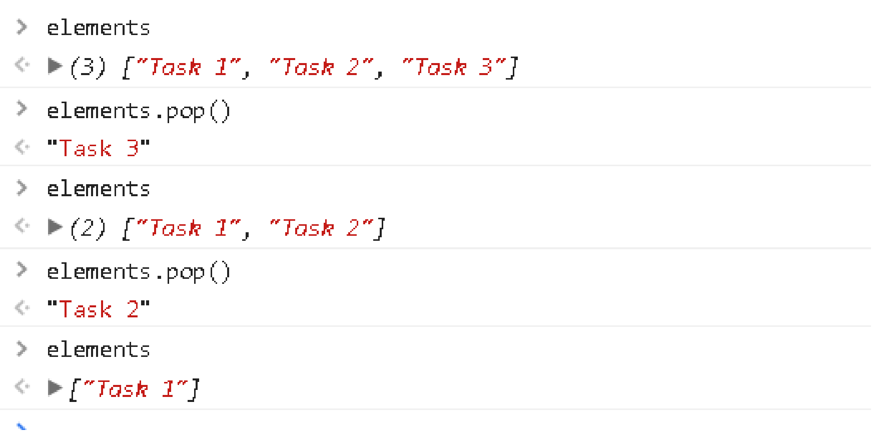
An array containing the deleted elements. If only one element is removed, an array of one element is returned. If no elements are removed, an empty array is returned.

```
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']
```

Remove last item from the array -> pop() method

The `pop()` method removes the **last** element from an array and returns that element. This method changes the length of the array.

```
var elements = ['Task 1', 'Task 2', 'Task 3'];
```



```
> elements
<· ▶(3) ["Task 1", "Task 2", "Task 3"]
> elements.pop()
<· "Task 3"
> elements
<· ▶(2) ["Task 1", "Task 2"]
> elements.pop()
<· "Task 2"
> elements
<· ▶["Task 1"]
```

pop() in action

Merging two arrays -> concat() method

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.

```
let array1 = ['a', 'b', 'c'];
let array2 = ['d', 'e', 'f'];

let merged = array1.concat(array2);

console.log(merged);
// expected output: Array ["a", "b", "c", "d", "e", "f"]
```

Merging two arrays -> es6 (spread operator)

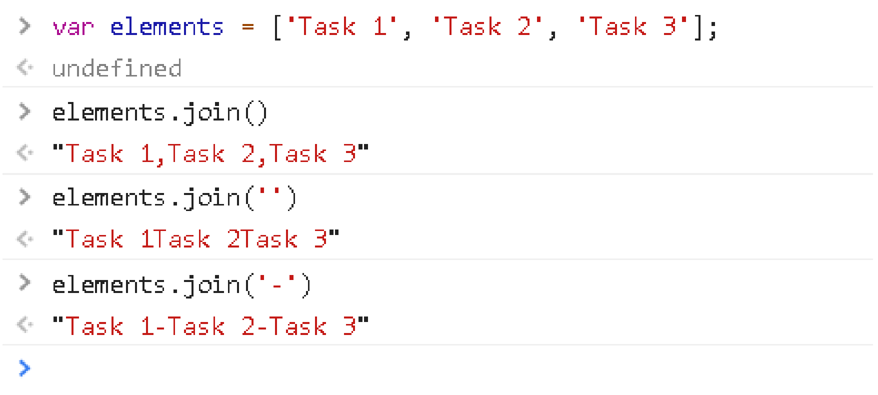
```
let array1 = ['a', 'b', 'c'];
let array2 = ['d', 'e', 'f'];

let merged = [...array1, ...array2];

console.log(merged);
// expected output: Array ["a", "b", "c", "d", "e", "f"]
```

Joining arrays -> Join () method

The `join()` method joins all elements of an array (or an array-like object) into a string and returns this string. This is a very useful method.

```
var elements = ['Task 1', 'Task 2', 'Task 3'];

console.log(elements.join());
// expected output: Task 1,Task 2,Task 3

console.log(elements.join(''));
// expected output: Task 1Task 2Task3

console.log(elements.join('-'));
// expected output: Task 1-Task 2-Task 3
```

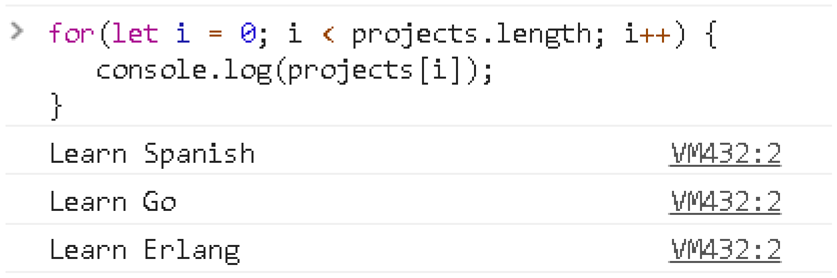
The output



```
> var elements = ['Task 1', 'Task 2', 'Task 3'];
<· undefined
> elements.join()
<· "Task 1,Task 2,Task 3"
> elements.join('')
<· "Task 1Task 2Task 3"
> elements.join('-')
<· "Task 1-Task 2-Task 3"
>
```

Looping through array

There are various ways to loop through array. Let's see the simple example first.



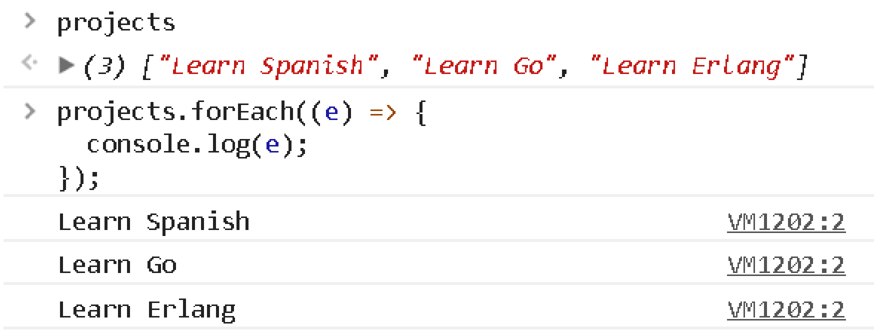
```
> for(let i = 0; i < projects.length; i++) {
      console.log(projects[i]);
  }
  Learn Spanish                                    VM432:2
  Learn Go                                         VM432:2
  Learn Erlang                                     VM432:2
```

for loop

Looping through array—forEach Loop

The forEach loop takes a function, a normal or arrow and gives access to individual element as parameter to the function. It takes two parameters, the first is the array element and the second is the index.

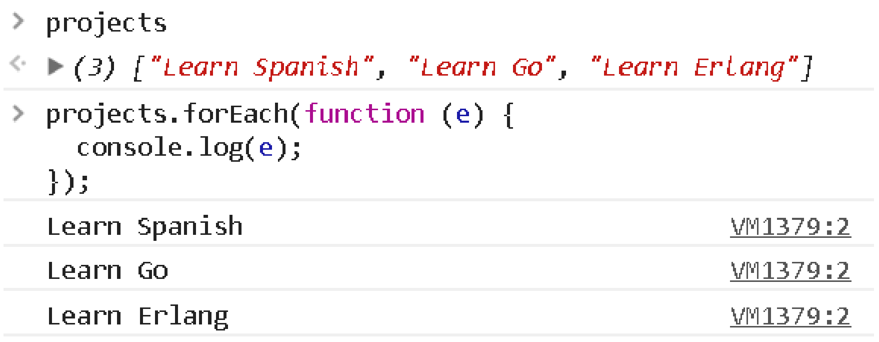
```
projects.forEach((e) => {
  console.log(e);
});
```



```
> projects
<· ▶(3) ["Learn Spanish", "Learn Go", "Learn Erlang"]
> projects.forEach((e) => {
    console.log(e);
  });
  Learn Spanish                                    VM1202:2
  Learn Go                                         VM1202:2
  Learn Erlang                                     VM1202:2
```

forEach (arrow function)

```
projects.forEach(function (e) {
  console.log(e);
});
```

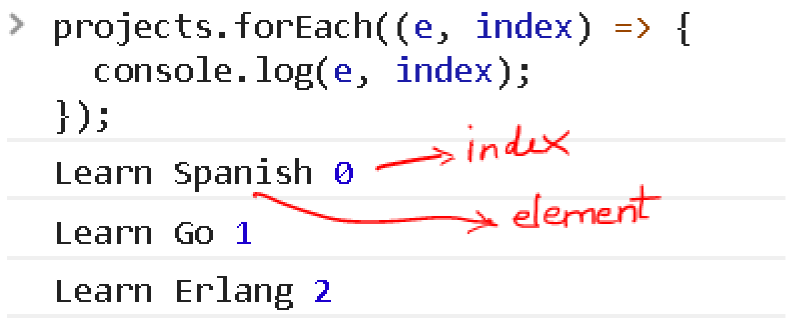


```
> projects
<· ▶(3) ["Learn Spanish", "Learn Go", "Learn Erlang"]
> projects.forEach(function (e) {
    console.log(e);
  });
  Learn Spanish                                    VM1379:2
  Learn Go                                         VM1379:2
  Learn Erlang                                     VM1379:2
```

forEach (normal function)

Let's see how we can access the index in forEach. Below I am using the arrow function notation, but will work for es5 function type as well.

```
projects.forEach((e, index) => {
  console.log(e, index);
});
```



```
> projects.forEach((e, index) => {
    console.log(e, index);
  });
  Learn Spanish 0
  Learn Go 1
  Learn Erlang 2
```

forEach with element and index parameter

Finding Elements in an array—find method

The `find()` method returns the **value** of the **first element** in the array that satisfies the provided testing function. Otherwise `undefined` is returned.

The syntax is given below.

- callback—Function to execute on each value in the array, taking three arguments

- ***** element—The current element being processed in the array
  ***** index (optional)—The index of the current element
  ***** array (optional)—The array find was called upon.
- thisArg (optional)—Object to use as `this` when executing callback.

Return value
A value in the array if any element passes the test; otherwise, undefined.

```
arr.find(callback[, thisArg])

var data = [51, 12, 8, 130, 44];

var found = data.find(function(element) {
  return element > 10;
});

console.log(found);  // expected output: 51
```

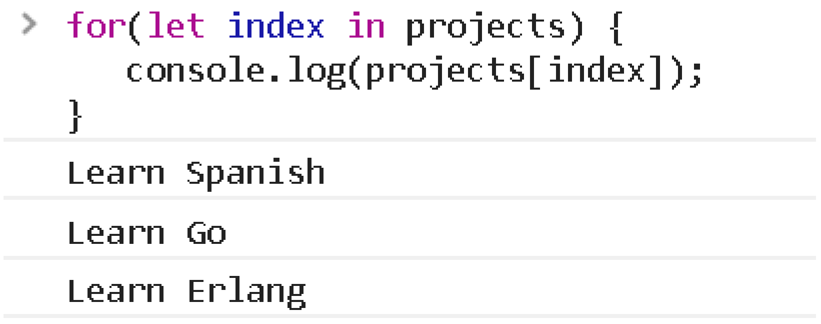
Looping through array—for in Loop

A `for...in` loop only iterates over enumerable properties and since arrays are enumerable it works with arrays.

The loop will iterate over all enumerable properties of the object itself and those the object inherits from its constructor's prototype (properties closer to the object in the prototype chain override prototypes' properties).

More reading at MDN for in loop

```
for(let index in projects) {
   console.log(projects[index]);
}
```



```
> for(let index in projects) {
     console.log(projects[index]);
  }
  Learn Spanish
  Learn Go
  Learn Erlang
```

Note in the above code, a new index variable is created every time in the loop.

Looping through array—map function ()

The map() function allows us to transform the array into new object and returns a new array based on the provided function.

It is a very powerful method in the hands of the JavaScript developer.

NOTE: Map always returns the same number of output, but it can modify the type of the output. For example if the array contains 5 element map will always return 5 transformed element as the output.

```
let num = [1,2,3,4,5];

let squared = num.map((value, index, origArr) => {
   return value * value;
});
```

The function passed to map can take three parameters.

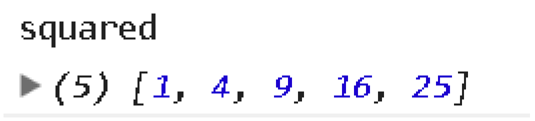
- `squared` —the new array that is returned
- `num` —the array to run the map function on
- `value` —the current value being processed
- `index` —the current index of the value being processed
- `origArr` —the original array

Map ()—Example 1—Simple

```
let num = [1,2,3,4,5];

let squared = num.map((e) => {
   return e * e;
});

console.log(squared);
```

In the above code we loop through all element in the array and create a new array with the square of the original element in the array.

A quick peek into the output.



```
squared
▶ (5) [1, 4, 9, 16, 25]
```

map -> simple input -> squared output

Map ()—Example 2— Simple Transformation

Let us take an input object literal and transform it into key value pair.

For example, let’s take the below array

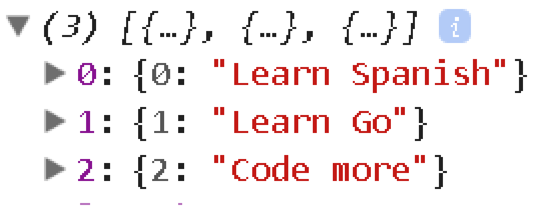
```
let projects = ['Learn Spanish', 'Learn Go', 'Code more'];
```

and transform into key value pair as shown below.

```
{
  0:  "Learn Spanish",
  1:  "Learn Go",
  2:  "Code more"
}
```

Here is the code for the above transformation with output.

```
let newProjects = projects.map((project, index) => {
   return {
     [index]: project
   }
});
console.log(newProjects);
```



```
▼ (3) [{…}, {…}, {…}]
  ▶ 0: {0: "Learn Spanish"}
  ▶ 1: {1: "Learn Go"}
  ▶ 2: {2: "Code more"}
```

Map ()—Example 3—Return a subset of data

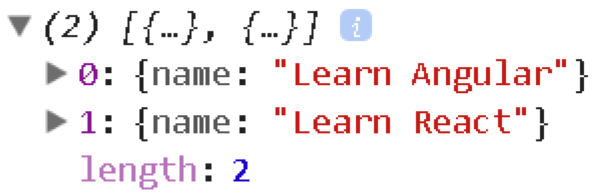
Lets take the below input

```
let tasks = [
   { "name": "Learn Angular",
     "votes": [3,4,5,3]
   },
   { "name": "Learn React",
     "votes": [4,4,5,3]
   },
];
```

The output that we need is just the name of the tasks. Let’s look at the implementation

```
let taskTitles = tasks.map((task, index, origArray) => {
   return {
     name: task.name
   }
});


console.log(taskTitles);
```

And here is the output.



```
▼ (2) [{…}, {…}]
  ▶ 0: {name: "Learn Angular"}
  ▶ 1: {name: "Learn React"}
    length: 2
```

Looping through array—filter function ()

Filter returns a subset of an array. It is useful for scenarios where you need to find records in a collection of records. The callback function to filter must return true or false. Return true includes the record in the new array and returning false excludes the record from the new array.

It gives a new array back.

Let's consider the below array as an input.

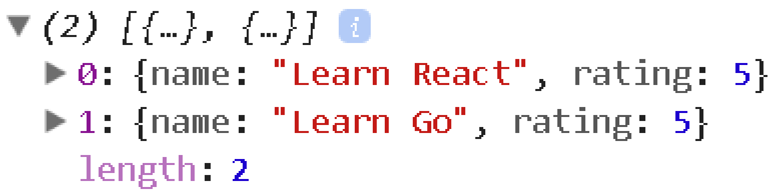
```
let tasks = [
   { "name": "Learn Angular",
     "rating": 3
   },
   { "name": "Learn React",
     "rating": 5
   },
   { "name": "Learn Erlang",
     "rating": 3
   },
   { "name": "Learn Go",
     "rating": 5
   },


];
```

Now lets use the `filter` function to find all tasks with a `rating` of 5.

Let's peek into the code and the result.

```
let tasks5 = tasks.filter((task) => {
    return task.rating === 5;
});
console.log(tasks5);
```

And the output is shown below.



```
▼(2) [{…}, {…}]
 ▶0: {name: "Learn React", rating: 5}
 ▶1: {name: "Learn Go", rating: 5}
  length: 2
```

Since we are only using one statement in the filter function we can shorten the above function as shown below.

```
tasks.filter(task => task.rating === 5);
```

NOTE: Filter function cannot transform the output into a new array.

Looping through array—reduce function ()

Reduce function loops through array and can result a reduced set. It is a very powerful function, I guess, more powerful than any other array methods (though every method has its role).

From MDN, The `reduce()` method applies a function against an accumulator and each element in the array (from left to right) to reduce it to a single value.

Reduce—Simple Example—1

```
const array1 = [1, 2, 3, 4];
const reducer = (accumulator, currentValue) => accumulator +
currentValue;

// 1 + 2 + 3 + 4
console.log(array1.reduce(reducer)); // expected output: 10

// 5 + 1 + 2 + 3 + 4
console.log(array1.reduce(reducer, 5));
// expected output: 15
```

Note: The first time the callback is called, `accumulator` and `currentValue` can be one of two values. If `initialValue` is provided in the call to `reduce()` , then `accumulator` will be equal to `initialValue` , and `currentValue` will be equal to the first value in the array. If no `initialValue` is provided, then `accumulator` will be equal to the first value in the array, and `currentValue` will be equal to the second.

The reason reduce is very powerful is because just with reduce() we can implement our own, map(), find() and filter() methods.

Using reduce to categorize data

Assume you have the below data structure which you would like to categorize into male and female dataset.

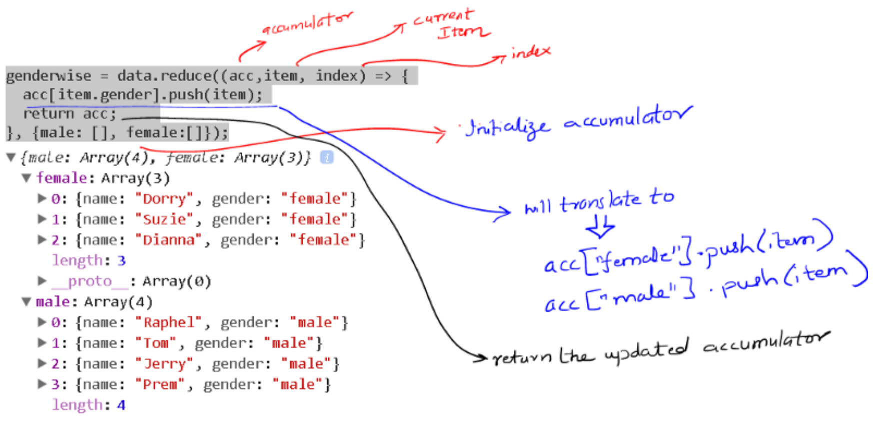
```
let data = [
  {name: "Raphel", gender: "male"},
  {name: "Tom", gender: "male"},
  {name: "Jerry", gender: "male"},
  {name: "Dorry", gender: "female"},
  {name: "Suzie", gender: "female"},
  {name: "Dianna", gender: "female"},
  {name: "Prem", gender: "male"},
];
```

And we would like the output to be as below

```
{"female" : [
    {name: "Dorry", gender:"female"},
    {name: "Suzie", gender: "female"},
    {name: "Dianna", gender: "female"},
 ],
 "male" : [
    {name: "Raphel", gender:"male"},
    {name: "Tom", gender:"male"},
    {name: "Jerry", gender:"male"},
    {name: "Prem", gender:"male"},
 ]
}
```

So, lets get to the code and understand how to achieve the above categorization.

```
let genderwise = data.reduce((acc,item, index) => {
  acc[item.gender].push(item);
  return acc;
}, {male: [], female:[]});

console.log(genderwise);
```



```
genderwise = data.reduce((acc,item, index) => {
  acc[item.gender].push(item);
  return acc;
}, {male: [], female:[]});
▼{male: Array(4), female: Array(3)}
  ▼female: Array(3)
    ▶0: {name: "Dorry", gender: "female"}
    ▶1: {name: "Suzie", gender: "female"}
    ▶2: {name: "Dianna", gender: "female"}
     length: 3
    ▶__proto__: Array(0)
  ▼male: Array(4)
    ▶0: {name: "Raphel", gender: "male"}
    ▶1: {name: "Tom", gender: "male"}
    ▶2: {name: "Jerry", gender: "male"}
    ▶3: {name: "Prem", gender: "male"}
     length: 4
```

Working of reduce for categorization

The important point above is the reduce function can be initialized with any type of starting accumulator, in the above case an object literal containing

```
{ male: [], female: []}
```

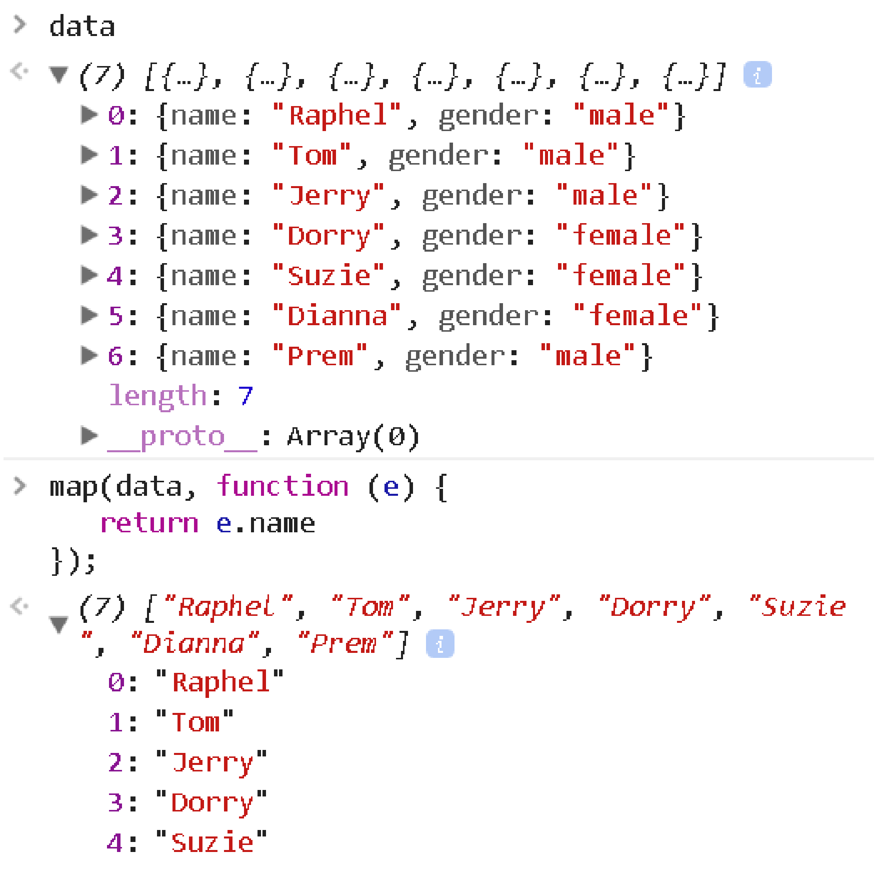
I hope this is sufficient to demonstrate the power of reduce.

Using reduce to implement custom map() function

```
function map(arr, fn) {
  return arr.reduce((acc, item) => [...acc, fn(item)], []);
}
```

The above is the implementation of custom map function. I n the above function we are passing empty array [] as the initial value for the accumulator and the reduce function is returning a new array with the values from the accumulator spread out and appending the result of invoking the callback function with the current item.

Lets see the usage and the output.



```
> data
<· ▼(7) [{…}, {…}, {…}, {…}, {…}, {…}, {…}]
    ▶0: {name: "Raphel", gender: "male"}
    ▶1: {name: "Tom", gender: "male"}
    ▶2: {name: "Jerry", gender: "male"}
    ▶3: {name: "Dorry", gender: "female"}
    ▶4: {name: "Suzie", gender: "female"}
    ▶5: {name: "Dianna", gender: "female"}
    ▶6: {name: "Prem", gender: "male"}
     length: 7
    ▶__proto__: Array(0)
> map(data, function (e) {
      return e.name
  });
<· ▼(7) ["Raphel", "Tom", "Jerry", "Dorry", "Suzie", "Dianna", "Prem"]
     0: "Raphel"
     1: "Tom"
     2: "Jerry"
     3: "Dorry"
     4: "Suzie"
```

Custom map ()

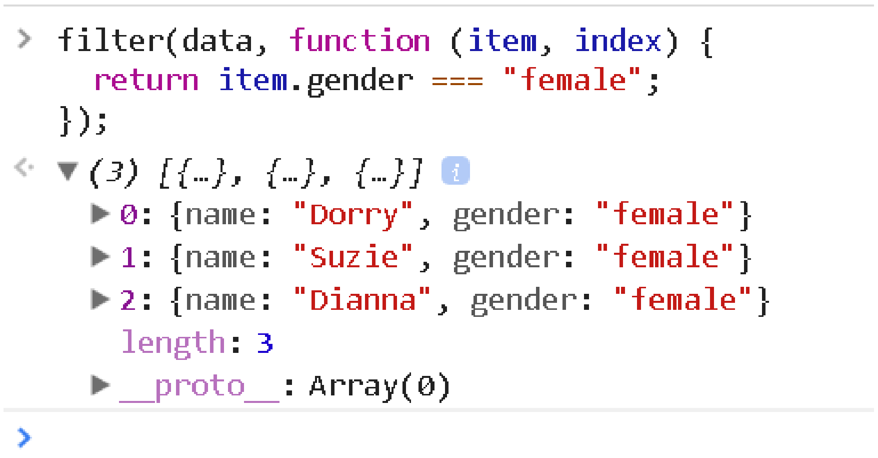
Using reduce to implement custom filter() function

Let’s implement the filter() method using reduce().

```
function filter (array, fn) {
  return arr.reduce(function (acc, item, index) {
    if (fn(item, index)) {
      acc.push(item);
    }
    return acc;
  },[]);
}
```

Let’s see the usage and the output below. I could have overwritten the original Array.prototype.filter, but am doing so to avoid manipulating built in methods.

Inside our custom filter, we are invoking the reduce function and only adding those items to the accumulator which matches the predicate that the callback function to filter returns.



```
> filter(data, function (item, index) {
    return item.gender === "female";
  });
<· ▼(3) [{…}, {…}, {…}]
    ▶0: {name: "Dorry", gender: "female"}
    ▶1: {name: "Suzie", gender: "female"}
    ▶2: {name: "Dianna", gender: "female"}
     length: 3
    ▶__proto__: Array(0)
>
```

Custom filter() using reduce

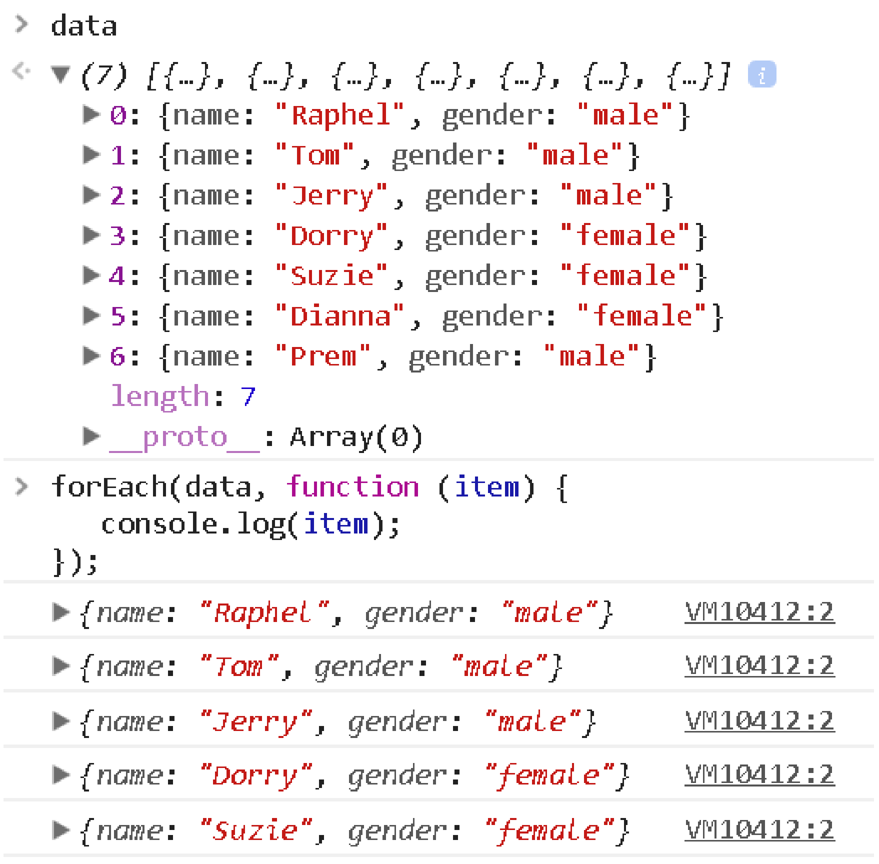
Using reduce to implement custom forEach function

Let us know implement our own forEach function.

```
function forEach(arr, fn) {
  arr.reduce((acc, item, index) => {
    item = fn(item, index);
  }, []);
}
```

The implementation is very simple compared to other methods. We just grab the passed in array, and invoke the reduce, and return the current item as a result of invoking the callback with the current item and index.

Let us see the usage and the output.



```
> data
<· ▼(7) [{…}, {…}, {…}, {…}, {…}, {…}, {…}]
   ▶0: {name: "Raphel", gender: "male"}
   ▶1: {name: "Tom", gender: "male"}
   ▶2: {name: "Jerry", gender: "male"}
   ▶3: {name: "Dorry", gender: "female"}
   ▶4: {name: "Suzie", gender: "female"}
   ▶5: {name: "Dianna", gender: "female"}
   ▶6: {name: "Prem", gender: "male"}
    length: 7
   ▶__proto__: Array(0)
> forEach(data, function (item) {
      console.log(item);
  });
  ▶{name: "Raphel", gender: "male"}      VM10412:2
  ▶{name: "Tom", gender: "male"}         VM10412:2
  ▶{name: "Jerry", gender: "male"}       VM10412:2
  ▶{name: "Dorry", gender: "female"}     VM10412:2
  ▶{name: "Suzie", gender: "female"}     VM10412:2
```

Custom forEach()

Holes in arrays

Holes in arrays means there are empty elements within the array. This may be because of couple of operations like delete or other operations that left these holes accidentally.

Now having 'holes' in array is not good from performance perspective. Let us take an example below.

```
let num = [1,2,3,4,5];  // No holes or gaps

delete num[2];  // Creates holes

console.log (num);  [1, 2, empty, 4, 5]
```

So, do not use `delete` method on array, unless you know what you are doing. `delete` method doesn't alter the length of the array.

You can avoid holes in array by using the array methods splice(), pop() or shift() as applicable.

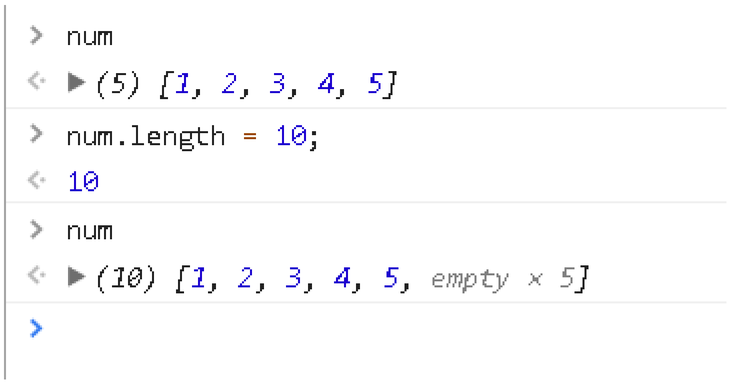
Changing array length and holes

You can quickly change the length of the array as below.

```
let num = [1,2,3,4,5];  // length = 5;
num.length = 3; // change length to 3


//The below logs outputs
// [1,2,3] -> The last two elements are deleted
console.log(num);
```

Now, increasing the length this way creates holes.

```
let num = [1,2,3,4,5];


num.length = 10;  // increase the length to 10


console.log(num);  // See holes here
```



```
> num
<· ▶(5) [1, 2, 3, 4, 5]
> num.length = 10;
<· 10
> num
<· ▶(10) [1, 2, 3, 4, 5, empty × 5]
>
```

The last 5 elements are holes.

Watch out this section for more details and tips.

Coming up

- Further updates

- Applications (use cases)

History

- Updated array holes and things to be careful
- Created the first draft

Conclusion

NOTE: Please note, this is a work in progress article and will be updated periodically.

Promotion: Special 10$ coupon for medium readers for my upcoming live ReactJS-Beyond the basics course on udemy.

codeburst.io

> 📨 Subscribe to *CodeBurst's* once-weekly **Email Blast**, 🐦 Follow *CodeBurst* on **Twitter**, view 🗺️ **The 2018 Web Developer Roadmap**, and 🕸️ **Learn Full Stack Web Development**.