

What is the JavaScript `reduce` Function?

The textbook definition of JavaScript `reduce` is:

JavaScript `reduce` is a higher order JavaScript function used for manipulating data that reduces an array to a single value.

But what this definition doesn't tell you is that this "single value" can be anything: number, string, array, or object. And that definition is what kept me from using it for a long time unless I needed to do something like get the sum of an array of numbers.

The `reduce` method of JavaScript runs a callback function on every element in an array. In this function, you use what is called an "accumulator" that you have complete control over. Just think of it as another variable where you collect everything you need from the original array.

JavaScript `reduce` Syntax

Here is the syntax of JavaScript `reduce`:

```
reduce(callbackFunction, initialValue)
```

The `initialValue` is optional, but you will want to set it most of the time, unless, for example, you are just summing an array of numbers or concatenating an array of strings. If you don't add the `initialValue`, which becomes the `accumulator` mentioned in the last section, the first element of your array (`myArray[0]`) will be used as the initial value and iteration through the array will start on the next element (`myArray[1]`).

Now let's look at the syntax for the callback function:

```
callbackFunction(accumulator, currentValue, currentIndex, array)
```

Here is what each of these arguments are:

- **accumulator:** When the function is first called this is the `initialValue` you added as the second argument of the `reduce` method. If there is no `initialValue`, it is the first element of the array being reduced. As the function is run on the array, you can modify this accumulator and the modified accumulator gets passed back in.
- **currentValue:** This is the current element of the array that the function is acting on. If you used an `initialValue`, the first call will be on `array[0]`. If not, it will be `array[1]`.
- **currentIndex:** This is the index of the current element of the array that the function is acting on.
- **array:** This is the full value of the array that the `reduce` function is acting on.

So, basically, when using `reduce`, you have access to just about information you need about the array, though in most instances, you will only have to use the first two parameters of the callback function. And when the `reduce` function has acted on every element in the array, it returns the final state of the `accumulator`.

How to Use `reduce` in JavaScript

Before we get into specific examples, let's look at one simple example using `reduce` without an initial value. This code simply takes an array that contains the alphabet and concatenates it into a string:

```
const alphaArray = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l',  
  
const alphaString = alphaArray.reduce((accumulator, currentValue) => {  
  return accumulator + currentValue;  
});  
  
console.log(alphaString);
```

The `reduce` function loops over the `alphaArray` and concatenates the next value to the accumulator. The result of this code will be:

```
abcdefghijklmnopqrstuvwxyz
```

In the next example, we are going to use `reduce` with an initial value. Here we have two arrays: one with the keys of the object we want to create and one with the values.

```
const keys = ['name', 'age', 'city'];  
  
const values = ['John Doe', 25, 'New York'];  
  
const result = keys.reduce((obj, key, index) => {  
  obj[key] = values[index];  
  return obj;  
}, {});
```

```
console.log(result);
```

And here is the result:

```
{  
  age: 25,  
  city: "New York",  
  name: "John Doe"  
}
```

In this use of `reduce`, we used different argument names for our callback function to better explain what values are being acted on.

- `accumulator` = `obj` (the initial value is an empty object)
- `currentValue` = `key` (from the `keys` array that is being reduced)
- `currentIndex` = `index`

And you should be able to see some of the magic that is possible with JavaScript `reduce`.

Our initial value is an empty object (`{}`) and it gets built up using these two arrays. But we only have to use `reduce` on one and then we can use the index of the current value in that array to get the value of the element with the same index in the other array to create the object.

While this might seem like a toy example, think of parsing CSV into a JSON object with the `keys` array being the first row in the CSV and it just gets more

complicated after that, but we will look at that and other specific examples in the next section.

JavaScript `reduce` Examples

Like I was saying earlier, you can do just about anything you want to do with an array using `reduce`. Here are some examples.

Using JavaScript `reduce` to Sum

When I first learned about `reduce`, this was all I thought it could do, so let's look at this example first:

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

const sum = numbers.reduce((accumulator, currentValue) => {
  return accumulator + currentValue;
}, 0);

console.log(sum); // result: 15
```

To begin, we set the `initialValue` of the `accumulator` to `0`, so we can add the numbers in the array to it.

Then, for each `currentValue` in the `numbers` array, we add it to the `accumulator`.

When the iterations are done, the final result is stored in the `sum` variable.

Using JavaScript `reduce` on an Array of Objects

Using `reduce` on an array of objects is not much different from using it on an array of values. Here is an example where our array contains a shopping list with the `name`, `price` and `quantity` of each item purchase and we want a total:

```
const products = [
  { name: 'Apple', price: 0.5, quantity: 10 },
  { name: 'Banana', price: 0.25, quantity: 5 },
  { name: 'Orange', price: 0.3, quantity: 8 }
];

const totalCost = products.reduce((accumulator, product) => {
  return accumulator + product.price * product.quantity;
}, 0);

console.log(totalCost); // result: 7.9
```

Inside the `reduce` method, we start with an initial value of `0` for the `accumulator`. Then, for each `product` object in the `products` array, we compute the cost by multiplying the `price` and `quantity` properties, and add it to the `accumulator`.

The `reduce` method iterates over each object in the `products` array, continuously accumulating the total cost.

Using JavaScript `reduce` on an Object

What? Yes, I know `reduce` is for arrays, but `Object.keys()` returns an array of the keys in an object. Here is how we can use this:

```
const person = {
  name: 'John',
  age: 30,
  city: 'New York'
};

const transformedObject = Object.keys(person).reduce((accumulator, key) => {
  accumulator[key.toUpperCase()] = person[key];
  return accumulator;
}, {});

console.log(transformedObject);
/*
Result
-----
{
  AGE: 30,
  CITY: "New York",
  NAME: "John"
}
*/
```

In the code above, we have an object and we want to convert the keys to uppercase.

Inside the `reduce` method, we use `Object.keys(person)` to get an array of keys from the `person` object. This allows us to iterate over each key.

For each key, we assign the corresponding value from `person[key]` to a new property in the `accumulator` object with the key converted to uppercase (`key.toUpperCase()`).

The `reduce` method iterates over each key in the `person` object, building up the `accumulator` object with the transformed properties.

Using JavaScript `reduce` to Flatten an Array of Arrays

Here is how you flatten an array of arrays into a single level array using

`reduce`

```
const nestedArray = [[1, 2], [3, 4], [5, 6]];

const flattenedArray = nestedArray.reduce((accumulator, currentArray) => {
  return accumulator.concat(currentArray);
}, []);

console.log(flattenedArray); // result [1, 2, 3, 4, 5, 6]
```

Setting our initial value to an empty array, we simply return the `accumulator` after concatenating the `currentArray` in the iteration. These arrays didn't have duplicates, but if they did, the next section will show you how to remove them.

Using JavaScript `reduce` to Remove Duplicates in an Array

It's easy to remove duplicates in an array with `reduce`. Here's how:

```
const numbers = [1, 2, 3, 4, 4, 5, 2, 6, 3, 7];

const uniqueNumbers = numbers.reduce((accumulator, currentValue) => {
  return accumulator.includes(currentValue)
    ? accumulator
    : [...accumulator, currentValue] ;
}, []);

console.log(uniqueNumbers); // result [1, 2, 3, 4, 5, 6, 7]
```


Inside the `reduce` method, we use an empty array (`[]`) as the initial value for the `accumulator`. Then, for each element (`currentValue`) in the `numbers` array, we check if it is already present in the `accumulator` using the `includes` method. If the element is not present, we return a new array, spreading the `accumulator` and adding the `currentValue` to it.

That example shows the functionality of `reduce`, but this is more effective:

```
const numbers = [1, 2, 3, 4, 4, 5, 2, 6, 3, 7];

const uniqueNumbers = [...new Set(numbers)];

console.log(uniqueNumbers); // result [1, 2, 3, 4, 5, 6, 7]
```

Using JavaScript `reduce` to Reverse an Array

You can also reverse an array using `reduce`:

```
const array = [1, 2, 3, 4, 5];

const reversedArray = array.reduce((accumulator, currentValue) => {
  return [currentValue, ...accumulator];
}, []);

console.log(reversedArray); // result: [5, 4, 3, 2, 1]
```

Here, we add the `currentValue` at the beginning of the array and then use the spread operator to put the current contents of the `accumulator` (the array we are building) at the end.

Using JavaScript `reduce` for Array Intersection

You can use JavaScript's `reduce` method to find all elements that are common in a list of arrays. Here's how:

Open in app ↗

Sign up

Sign in

Medium

🔍 Search

✍ Write



```
[3, 4, 5, 8]
];

const intersection = arrays.reduce((accumulator, currentArray) => {
  return accumulator.filter((element) => currentArray.includes(element));
});

console.log(intersection); // result [4, 5]
```

Here we use the `accumulator` to collect all the common values. Then we use the JavaScript `filter` method to check if each element in the `accumulator` is present in the `currentArray` using the `includes` method. Only elements that fit this condition are kept in the `accumulator`.

Using JavaScript `reduce` to Create a Pipeline of Functions

If you want to run a bunch of different functions on a value, you can create those functions separately and then use `reduce` to create a pipeline that runs them in order. Here is an example:

```
const addOne = (a) => a + 1;
const multiplyByTwo = (a) => a * 2;
const divideByThree = (num) => num / 3;

const pipeline = [addOne, multiplyByTwo, divideByThree];
```

```
const result = pipeline.reduce(function(total, func) {  
  return func(total);  
}, 8);  
  
console.log(result); // result: 6
```

Here, the value we are modifying is set at the initial value, the array contains our functions, and we iterate over them, we run them on the `initialValue/accumulator`. If we wanted to, we could wrap the method in another function and pass in the value we want as the `initialValue`. We could even take it further and turn this new function into a callback for another `reduce` and apply all three functions to all elements in another array.

Using JavaScript `reduce` to Convert CSV into a JSON Object

Earlier in the article, I mentioned showing how to convert a CSV document into JSON. The only thing I don't have in this code is reading the file:

```
const csvData = `Name,Age,City  
John,25,New York  
Jane,30,San Francisco  
Alex,35,Seattle`;  
  
const rows = csvData.split('\n');  
const headers = rows[0].split(',');  
  
const jsonData = rows.slice(1).reduce((accumulator, row) => {  
  const values = row.split(',');  
  const obj = headers.reduce((objAccumulator, header, index) => {  
    objAccumulator[header] = values[index];  
    return objAccumulator;  
  }, {});  
  
  return [...accumulator, obj];  
}, []);
```

```
console.log(jsonData); // result is next code block
```

First, we do some data prep and split the `csvData` string into an array of rows using the newline (`\n`) character as the delimiter. We also split the first row (header row) into an array of headers, so that it can be used during the process to set the object keys.

Inside the `reduce` method, we start with an empty array (`[]`) as the initial value for the `accumulator`. Then, for each row (starting from index 1 to exclude the header row), we split the row into an array of values using the comma (`,`) as the delimiter.

Inside the nested `reduce` method, we iterate over the `headers` array. For each header, we assign the corresponding value from the `values` array to the corresponding property in the `obj` object. Notice that this `reduce` methods `initialValue` is an empty object (`{}`).

This is a use case for the `currentIndex` parameter in a `reduce` callback. Since the header array length and each of the rows are the same size, we use the `index` of the current `headers` element to find the corresponding value in the row.

After processing all headers and values, we return a new array where we spread the `accumulator` array and add the `obj` object.

And the result:

```
[{
  Age: "25",
  City: "New York",
  Name: "John"
}, {
  Age: "30",
  City: "San Francisco",
  Name: "Jane"
}, {
  Age: "35",
  City: "Seattle",
  Name: "Alex"
}]
```

Replacing JavaScript `map`, `filter`, and `find` with `reduce`

You can also use JavaScript `reduce` to replace these other array methods. Why would you want to do that? Because you might want to do multiple things in one pass over the array instead of running multiple functions in multiple passes over the array. This shows you how to recreate those with `reduce` so you can add more functionality. But again, if all you need is `map`, `filter`, or `find`, you don't want to replace them with `reduce`.

Replacing JavaScript `map` with `reduce`

If you are wondering, “JavaScript `reduce` vs `map`, which should I use”. First, `map` can only return an array of the same size. You can manipulate all the values in the array, but the return will be the same length. Here is an example of using JavaScript `map` to double the value of each number in an array:

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

const doubledNumbers = numbers.map((num) => num * 2);
```

```
console.log(doubledNumbers); // result: [2, 4, 6, 8, 10]
```

The JavaScript `map` method creates a new array, multiplying each number in the array by 2.

And here we do the same thing with `reduce`:

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

const doubledNumbers = numbers.reduce((result, num) => {
  return [...result, num * 2];
}, []);

console.log(doubledNumbers); // result: [2, 4, 6, 8, 10]
```

Here, the accumulator parameter is named `result` and the current value is `num`. We start with an empty array as our initial value. The `reduce` method iterates over each element in the `numbers` array, doubles it, and adds the result to the `result` array. When it is done, our callback function returns the `result` array.

Replacing JavaScript `filter` with `reduce`

Unlike JavaScript `map`, JavaScript `filter` can return a shorter array than the original because it filters the array. Here is an example of using JavaScript `filter` to return an array with only even numbers:

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

const evenNumbers = numbers.filter((num) => num % 2 === 0);

console.log(evenNumbers); // result: [2, 4]
```

The `%` is the JavaScript remainder operator. It returns the remainder left over when one number is divided by another and even numbers, when divided by 2, have no remainder.

Here is how to replace JavaScript `filter` with `reduce` in this case:

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

const evenNumbers = numbers.reduce((result, num) => {
  return num % 2 === 0 ? [...result, num] : result;
}, []);

console.log(evenNumbers); // result: [2, 4]
```

This is similar to the `map` replacement example, except we only want to add even numbers to the resulting array. We use the remainder operator again here and when the number passes our even check, we add it to the array. If not, we just return the array as is.

Replacing JavaScript `find` with `reduce`

JavaScript `find` returns the first element in an array that passes the testing function. Below we want to find the first number in this array that is greater than 10.

```
const numbers = [5, 50, 8, 145, 3];

const found = numbers.find((num) => num > 10);

console.log(found); // result: 50
```

And here is how we can replace `find` with `reduce`:

```
const numbers = [5, 50, 8, 145, 3];

const found = numbers.reduce((result, num) => {
  if (result === undefined && num > 10) {
    result = num;
  }
  return result;
}, undefined);

console.log(found); // result: 50
```

The code above is different from the other examples. This is because `find` returns the first element matching our check. To do that with `reduce`, we set the initial value to `undefined`. We also check that it is still `undefined` before setting our result. If we didn't, it would return the last number in the array that passes our test or 3.

How to Break Out of JavaScript `reduce`

What you might notice when we replaced JavaScript `find` with JavaScript `reduce` in the last example, there is some wasted processing going on. After all, we found the number in the second element of the array, yet the code stills runs until it hits the end.

Unfortunately, there is no way to break out of the `reduce` method at this point. Well, there is, but not something you would really want to use. Here's how:

```
const numbers = [5, 50, 8, 145, 3];

try {
  const found = numbers.reduce((result, num) => {
    if (num > 10) {
      throw new Error(`${num} found. Breaking out of reduce.`);
    }
    return result + num;
  }, 0);
} catch (error) {
  console.log(error.message); // result: "50 found. Breaking out of reduce."
}
```

Like I said, not the best way, so with `reduce`, we are stuck with looping through the whole array.

JavaScript `reduce` with Async/Await

There are a few ways to do this, but my favorite is using `Promise.all`, since will execute all the promises in parallel. In this example, we want to fetch data from multiple APIs and return the results:

```
async function fetchDataInParallel(urls) {
  try {
    const promises = urls.reduce((accumulator, url) => {
      return [...accumulator, fetch(url).then(response => response.json())];
    }, []);

    return await Promise.all(promises);
  } catch (error) {
```

```

        console.error('Error fetching data:', error);
    }
}

const urls = [
    'https://random.dog/woof.json',
    'https://cataas.com/api/cats?tags=cute&limit=3'
];

fetchDataInParallel(urls)
    .then(data => console.log(data))
    .catch(error => console.error('Error:', error)); // result below

```

First, we create a reusable function that we can pass in our array of `urls` to hit. We add a `try...catch` block around the functions to handle potential error.

`const promises = urls.reduce(...)` creates an array of promises: ▶

- `[...accumulator, fetch(url).then(response => response.json())]` : Returns a new array with containing the elements of the accumulator and the new Promise.
- `return accumulator` : Returns the updated accumulator for the next iteration.
- `[]` : Initial accumulator is an empty array to store promises.

Then, `const results = await Promise.all(promises)` waits for all the promises to resolve and collects the results and the results are returned.

And here are the results. I don't know what you would use this data for, since I picked random public APIs, but the code is currently fully functional.

```
[{
  fileSizeBytes: 107720,
  url: "https://random.dog/e00b0661-9c04-463c-8f88-612613dd0ea0.jpeg"
}, [{
  _id: "3zM2HCQRdiZHB30o",
  mimetype: "image/jpeg",
  size: 22511,
  tags: ["calico", "sleepy", "cute", "fat", "luna"]
}, {
  _id: "aonUpTntYMPno4yt",
  mimetype: "image/jpeg",
  size: 38309,
  tags: ["fat", "tricolor", "cute", "calico"]
}, {
  _id: "zeAG7BfSGsweyeS0",
  mimetype: "image/jpeg",
  size: 36719,
  tags: ["cute", "face", "white"]
}]]
```

So What Is JavaScript `reduceRight` ?

This is simply JavaScript `reduce` that runs backwards. Instead of iterating from the beginning of the array, it starts at the end. The syntax is the same as `reduce`. Here is our first example using `reduceRight` instead of `reduce`:

```
const alphaArray = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l',
const alphaString = alphaArray.reduceRight((accumulator, currentValue) => {
  return accumulator + currentValue;
});

console.log(alphaString); // result: zyxwvutsrqponmlkjihgfedcba
```

And instead of concatenating the alphabet in order, it does it in reverse order.

Conclusion

In this article, I wanted to help people get over the hurdles of learning JavaScript `reduce`, because it is a really useful tool and its definition and syntax description aren't really enough to demonstrate its power. Hopefully, some of these examples helped you see how useful it can be. So, go forth and reduce some things. And for those of you who just want to point out the flaws, I have said multiple times, this is to demonstrate how `reduce` can be used and not where it should be used and you can instead go forth and be the best troll you can be.

Originally published at <https://www.stephanmiller.com> on January 11, 2024.

JavaScript

Js

Coding

Programming

Development

