Using the Array Reduce Method

- Due No Due Date
- Points 1
- Submitting a website url

[(https://github.com/learn-co-curriculum/phase-1-using-array-reduce)](https://github.com/learn-co-curriculum/phase-1-using-array-reduce) [(https://github.com/learn-co-curriculum/phase-1-using-array-reduce/issues/new)](https://github.com/learn-co-curriculum/phase-1-using-array-reduce/issues/new)

Learning Goals

- Learn how the `reduce()` method works
- Demonstrate using `reduce()` with a primitive return value
- Demonstrate using `reduce()` with an object as the return value

Introduction

In the world of programming, we often work with lists. Sometimes we want to find or transform elements in a list, but other times we might want to create a single summary value. In this lesson, we'll learn how to use the `reduce()` iterator method to **aggregate** a result, i.e., to *reduce* a list to a single value. That value can be a string, a number, a boolean, etc.

To better understand how `reduce()` works, we'll start by building our own version.

Learn How the `reduce()` Method Works

Let's say we have a bunch of grocery items in our basket and we want to calculate the total price. Our basket data might look like this:

```
const products = [
  { name: "Shampoo", price: 4.99 },
  { name: "Donuts", price: 7.99 },
  { name: "Cookies", price: 6.49 },
  { name: "Bath Gel", price: 13.99 },
];
```

We're going to *reduce* the array of products to a *single value*: the total price. To do this, we'll create a `getTotalAmountForProducts()` function:

```
function getTotalAmountForProducts(products) {
  let totalPrice = 0;

  for (const product of products) {
    totalPrice += product.price;
  }

  return totalPrice;
}

console.log(getTotalAmountForProducts(products)); // LOG: 33.46
```

We first declare a `totalPrice` variable and set its initial value to 0. We then iterate through the products in the basket and add the price of each to the total. When the loop has finished, we return the `totalPrice` .

This is a very basic way to manually add together the prices of the products we want to buy, but it only works for this very specific situation. We could make our solution more abstract by writing a generalized function that accepts two additional arguments: an initial value and a callback function that implements the reduce functionality we want.

To see what this might look like, let's count the number of coupons we have lying around the house:

```
const couponLocations = [
  { room: "Living room", amount: 5 },
  { room: "Kitchen", amount: 2 },
  { room: "Bathroom", amount: 1 },
  { room: "Master bedroom", amount: 7 },
];
```

```
function ourReduce(arr, reducer, init) {
  let accum = init;
  for (const element of arr) {
    accum = reducer(accum, element);
  }
  return accum;
}

function couponCounter(totalAmount, location) {
  return totalAmount + location.amount;
}

console.log(ourReduce(couponLocations, couponCounter, 0)); // LOG: 15
```

`ourReduce()` accepts three arguments: the array we want to reduce, the callback function or *reducer*, and the initial value for our *accumulator* variable. It then iterates over the array, calling the reducer function each time, which returns the updated value of the accumulator. The final value of the accumulator is returned at the end.

Note that `ourReduce()` is generalized: the specifics (the callback function and initial value) have been abstracted out, making our code more flexible. If, for example, we already have three coupons in our hand, we can easily account for that without having to change any code by adjusting the initial value when we call `ourReduce()` :

```
console.log(ourReduce(couponLocations, couponCounter, 3)); // LOG: 18
```

Demonstrate using `reduce()` with a Primitive Return Value

With JavaScript's native `reduce()` method, we don't need to write our own version. Just like `ourReduce` , the `reduce()` method is used when we want to get some information from each element in the collection and gather that information into a final summary value. Let's take the native implementation for a spin with our previous example:

```
console.log(couponLocations.reduce(couponCounter, 0)); // also logs 15!
```

Another simple numerical example:

```
const doubledAndSummed = [1, 2, 3].reduce(function (accumulator, element) {
  return element * 2 + accumulator;
}, 0);
// => 12
```

Here, as in the previous example, we are calling `.reduce()` on our input array and passing it two arguments: the callback function, and an optional start value for the accumulator (0 in this example). `.reduce()` executes the callback for each element in turn, passing in the current value of the accumulator and the current element each time. The callback updates the value of the accumulator in each iteration, and that updated value is then passed as the first argument to the callback in the next iteration. When there's nothing left to iterate, the final value of the accumulator (the total) is returned.

The initialization value is optional, but leaving it out might lead to a real surprise. If no initial value is supplied, the *first element in the array* is used as the starting value. `reduce()` then executes the callback function, passing this starting value and the *second* element of the array as the two arguments. In other words, the code inside the callback **is never executed** for the first element in the array. This can lead to unexpected results:

```
const doubledAndSummed = [1, 2, 3].reduce(function (accumulator, element) {
  return element * 2 + accumulator;
});
// => 11
```

In some cases, it won't matter (e.g., if our reducer is simply summing the elements of the input array). However, to be safe, it is best to always pass a start value when calling `reduce()` . Of course, that initial value can be anything we like:

```
const doubledAndSummedFromTen = [1, 2, 3].reduce(function (
  accumulator,
  element
) {
  return element * 2 + accumulator;
},
10);
// => 22
```

Demonstrate using `reduce()` with an Object as the Return Value

The output of the `reduce()` method does not need to be a primitive value like a `Number` or `String` . Let's consider a couple of examples that accumulates array values into an `Object` .

First, let's look at an example where we take an array of letters and return an object with letters as keys and their count in the array as values.

```
const letters = ["a", "b", "c", "b", "a", "a"];

const lettersCount = letters.reduce(function (countObj, currentLetter) {
  if (currentLetter in countObj) {
    countObj[currentLetter]++;
  } else {
    countObj[currentLetter] = 1;
  }
  return countObj;
}, {});

console.log(lettersCount); // { a: 3, b: 2, c: 1 }
```

We initialize the `countObj` as an empty object by passing `{}` as the second argument to the `reduce` method. The callback method increments the current letter's count in the `countObj` if it already exists otherwise, initializes it to `1` .

Let's look at a more complex example now. Say we want to create a roster of student artists based on their discipline of art for their final showcase. Our start value might look like this:

```
const artsShowcases = {
  Dance: [],
  Visual: [],
  Music: [],
  Theater: [],
  Writing: [],
};
```

Imagine we also have a `studentSorter` object that includes a `showcaseAssign()` method. That method takes the name of a student as its argument and returns the name of the showcase the student should be assigned to. Note that we have not coded out the `showcaseAssign()` method — the details of how it would work are not important for our purposes. What's important to remember is that the method takes the name of a student as an argument and returns one of the five showcases: "Dance", "Visual", "Music", "Theater" or "Writing". We want to call the method for each element in our input array (each student's name), get the value of the showcase that's returned, and add the student's name to the array for that showcase in the `artsShowcases` object.

To do that, we will call reduce on our input array, `incomingStudents` , which contains the names of all incoming students, passing a callback function and the start value of `artsShowcases` as the arguments. The callback is where we'll push each student name into the appropriate showcase:

```
incomingStudents.reduce(function (showcases, student) {
  showcases[studentSorter.showcaseAssign(student)].push(student);
}, artsShowcases);
```

Let's break this down: `.reduce()` executes the callback for each student name in turn. Inside the callback, the `studentSorter.showcaseAssign()` method is called with the current student name as its argument. `showcaseAssign()` returns the name of an Arts Showcase, which is then used as the key to access the correct array in the `artsShowcases` object and push the student's name into it. The iteration then continues to the next element in the array, passing the next student name and the updated value of `artsShowcases` as the arguments. Once `reduce()` has iterated through all the students in `incomingStudents` , it returns the final value of `artsShowcases` .

We can then access the list of students in any Arts Showcase:

```
artsShowcases["Visual"]; //=> [yishayGarbasz, wuTsang, mickaleneThomas]
```

Lab: Use `reduce()` to Create a Single Aggregate of All Items in a List

Let's say we are hard at work in the battery factory. We've assembled several batches of batteries today. Let's count how many assembled batteries we ended up with.

- Create a new variable called `totalBatteries` , which holds the sum of all of the battery amounts in the `batteryBatches` array. (Note that the `batteryBatches` variable has been provided for you in `index.js` .) Naturally, you should use `reduce()` for this!

Remember the workflow:

1. Install the dependencies using `npm install` .
2. Run the tests using `npm test` .
3. Read the errors; vocalize what they're asking you to do.
4. Write code; repeat steps 2 and 3 often until a test passes.
5. Repeat as needed for the remaining tests.

After you have all the tests passing, remember to commit and push your changes up to GitHub, then submit your work to Canvas using CodeGrade.

Conclusion

With `reduce()` , we are able to quickly get a single summary value from the elements in an array. `reduce()` — like the other iterator methods we've learned about in this section — can greatly cut down the amount of time spent recreating common functionality. It can also make our code more efficient and expressive.

Resources

- **MDN: Array.prototype.reduce() (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce)**