forEach: The Iterator of No Return

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

Learning Goals

- Use `forEach()` to work with an array
- Explain why `forEach()` is the "least expressive" iterator method
- Identify use cases for `forEach()`

Introduction

In previous lessons, we've learned about JavaScript's built-in `Array` methods and how they help us save work and write more efficient, readable code. In this lesson, we'll talk about one more: `forEach()` . We'll also discuss why it's the "least expressive" iterator.

Use `forEach` to Work with an Array

If you look at the MDN page for the `forEach()` method, you'll see the following description:

> The forEach() method executes a provided function once for each array element.

Unlike the other methods we've looked at in this section, `forEach()` doesn't have a built-in return value. As a result, `forEach()` is quite generic — the callback we pass to it can contain whatever functionality we like.

To use `forEach()` , we simply call it on an array and pass our callback:

```
oppressedWorkers = [
  "Dopey",
  "Sneezy",
  "Happy",
  "Angry",
  "Doc",
  "Lemonjello",
  "Sleepy",
];

oppressedWorkers.forEach(function (oppressedWorker) {
  console.log(`${oppressedWorker} wants to form a union!`);
}); //=> undefined

/* Output
Dopey wants to form a union!
Sneezy wants to form a union!
Happy wants to form a union!
Angry wants to form a union!
Doc wants to form a union!
Lemonjello wants to form a union!
Sleepy wants to form a union!
*/
```

While this flexibility may seem like a good thing at first glance, the fact that it's generic makes it the least expressive of the iterators.

Explain Why `forEach` is the Least Expressive Iterator Method

By now you recognize that `map` means: "create a new `Array` after transforming each element." You recognize that `reduce` means: "distill a single summary value from a set of elements." These methods are *expressive*; their presence in your code tells other programmers (and your future self) what you intended to happen.

But what does `forEach` mean? Programmers recognize that `map()` has a specific use, `reduce()` has a specific use, `find()` has a specific use. But `forEach()` is generic. Are we just printing things, or are we trying to distill to a value, or are we trying to produce a transformed `Array` ?

When we use `forEach()` to do `map` -things or `reduce` -things we're not *documenting* what our intention was with regard to the collection. This makes for code that's harder to understand and debug. Here's some code that uses `forEach()` instead of `reduce()` .

```
function sumArray(numberArray) {
  let total = 0;
  numberArray.forEach(function (i) {
    total = total + i;
  });
  return total;
}
sumArray([1, 2, 3]); //=> 6
```

Sure, it works, but it doesn't *communicate*. We should always strive to have code that works ***and*** communicates.

Identify Use Cases for `forEach`

The best time to use `forEach()` is when you need to enumerate a collection to cause some sort of "side-effect". A good example of this is when you want to iterate through an array to log values. `console.log()` doesn't return anything back, so using something like `map()` here would unnecessarily create a new array. It would also mislead any developers who look at your code about what its purpose is. We're using `forEach` strictly to do something that is handy for us (the developer) as a *side-effect*; in this case, printing content to the screen.

This is pretty common in debugging:

```
empCollection.forEach(function (e) {
  console.log("DEBUG: WHAT ARE YOU?!?" + e);
});
```

The other time we want to use `forEach` is if we need to directly change (or "mutate") the elements we're iterating through.

As an example, consider:

```
function addFullNameToEmployees(empCollection) {
  empCollection.forEach(function (e) {
    e.fullName = `${e.firstName} ${e.familyName}`;
  });
}

addFullNameToEmployees([
  { firstName: "Byron", familyName: "Karbitii" },
  { firstName: "Luca", familyName: "Tuexedensis" },
]);
```

In this case, we're directly updating employees in the original object, rather than creating a new object with the modifications. The employee, `e`, is updated as a *side-effect* of running `forEach`. The only clue that helps us guess what `forEach` is doing here is that the programmer "wrapped" it inside of a helpfully-named function.

Recall, however, that directly mutating objects is something that should be avoided in most cases. It's generally better to use the more expressive `map()` method for the use case above and save `forEach()` for cases where we aren't concerned about the return value.

Conclusion

In this lesson, we've introduced `Array.prototype.forEach()`. It is flexible and straightforward to use, which can make it an attractive option when you aren't sure which iterator is the best one for your needs. However, in the majority of cases, one of the other iterator methods will be a better choice. You should only use `forEach()` under the specific circumstances outlined above.

As you continue learning JavaScript, you should take advantage of the *expressive* iterator methods we've learned about in this section as much as possible. Under most circumstances, using `find()`, `filter()`, `map()`, or `reduce()` in place of more generic options, `forEach`, `for...of`, and `for`, will save you work in the long run and make your code more efficient and expressive.