# Introduction to Iterators

Imagine you had a grocery list and you wanted to know what each item on the list was. You'd have to scan through each row and check for the item. This common task is similar to what we have to do when we want to iterate over, or loop through, an array. One tool at our disposal is the for loop. However, we also have access to built-in array methods which make looping easier.

The built-in JavaScript array methods that help us iterate are called iteration methods, at times referred to as iterators. Iterators are methods called on arrays to manipulate elements and return values.

In this lesson, you will learn the syntax for these methods, their return values, how to use the documentation to understand them, and how to choose the right iterator method for a given task.

**The .forEach() Method**

The first iteration method that we're going to learn is .forEach(). Aptly named, .forEach()will execute the same code for each element of an array.

https://s3.amazonaws.com/codecademy-content/courses/learn-javascript-iterators/iterator+anatomy.svg

The code above will log a nicely formatted list of the groceries to the console. Let's explore the syntax of invoking .forEach().

* groceries.forEach() calls the forEachmethod on the groceries array.
* .forEach() takes an argument of callback function. Remember, a callback function is a function passed as an argument into another function.
* .forEach() loops through the array and executes the callback function for each element. During each execution, the current element is passed as an argument to the callback function.
* The return value for .forEach() will always be undefined.

Another way to pass a callback for .forEach() is to use arrow function syntax.

groceries.forEach(groceryItem => console.log(groceryItem));

We can also define a function beforehand to be used as the callback function.

function printGrocery(element){ console.log(element); } groceries.forEach(printGrocery);

The above example uses a function declaration but you can also use a function expression or arrow function as well.

All three code snippets do the same thing. In each array iteration method, we can use any of the three examples to supply a callback function as an argument to the iterator. It's good to be aware of the different ways to pass in callback functions as arguments in iterators because developers have different stylistic preferences. Nonetheless, due to the strong adoption of ES6, we will be using arrow function syntax in the later exercises.

**The .map() Method**

The second iterator we're going to cover is .map(). When .map() is called on an array, it takes an argument of a callback function and returns a new array! Take a look at an example of calling .map():

const numbers = [1, 2, 3, 4, 5]; const bigNumbers = numbers.map(number => { return number \* 10; });

.map() works in a similar manner to .forEach()— the major difference is that .map() returns a new array.

In the example above:

* numbers is an array of numbers.
* bigNumbers will store the return value of calling .map() on numbers.
* numbers.map will iterate through each element in the numbers array and pass the element into the callback function.
* return number \* 10 is the code we wish to execute upon each element in the array. This will save each value from the numbers array, multiplied by 10, to a new array.

If we take a look at numbers and bigNumbers:

console.log(numbers); // Output: [1, 2, 3, 4, 5] console.log(bigNumbers); // Output: [10, 20, 30, 40, 50]

Notice that the elements in numbers were not altered and bigNumbers is a new array.

**The .filter() Method**

Another useful iterator method is .filter(). Like .map(), .filter() returns a new array. However, .filter() returns an array of elements after filtering out certain elements from the original array. The callback function for the .filter() method should return trueor false depending on the element that is passed to it. The elements cause the callback function to return true are added to the new array. Take a look at the following example:

const words = ['chair', 'music', 'pillow', 'brick', 'pen', 'door']; const shortWords = words.filter(word => { return word.length < 6; });

* words is an array that contains string elements.
* const shortWords = declares a new variable that will store the returned array from invoking .filter().
* The callback function is an arrow function has a single parameter, word. Each element in the words array will be passed to this function as an argument.
* word.length < 6; is the condition in the callback function. Any word from the words array that has fewer than 6characters will be added to the shortWords array.

Let's also check the values of words and shortWords:

console.log(words); // Output: ['chair', 'music', 'pillow', 'brick', 'pen', 'door']; console.log(shortWords); // Output: ['chair', 'music', 'brick', 'pen', 'door']

Observe how words was not mutated, i.e. changed, and shortWords is a new array.

**The .findIndex() Method**

We sometimes want to find the location of an element in an array. That's where the .findIndex() method comes in! Calling .findIndex() on an array will return the index of the first element that evaluates to true in the callback function.

const jumbledNums = [123, 25, 78, 5, 9]; const lessThanTen = jumbledNums.findIndex(num => { return num < 10; });

* jumbledNums is an array that contains elements that are numbers.
* const lessThanTen = declares a new variable that stores the returned index number from invoking .findIndex().
* The callback function is an arrow function has a single parameter, num. Each element in the jumbledNums array will be passed to this function as an argument.
* num < 10; is the condition that elements are checked against. .findIndex() will return the index of the first element which evaluates to true for that condition.

Let's take a look at what lessThanTenevaluates to:

console.log(lessThanTen); // Output: 3

If we check what element has index of 3:

console.log(jumbledNums[3]); // Output: 5

Great, the element in index 3 is the number 5. This makes sense since 5 is the first element that is less than 10.

If there isn't a single element in the array that satisfies the condition in the callback, then .findIndex() will return -1.

const greaterThan1000 = jumbledNums.findIndex(num => { return num > 1000; }); console.log(greaterThan1000); // Output: -1

**The .reduce() Method**

Another widely used iteration method is .reduce(). The .reduce() method returns a single value after iterating through the elements of an array, thereby *reducing* the array. Take a look at the example below:

const numbers = [1, 2, 4, 10]; const summedNums = numbers.reduce((accumulator, currentValue) => { return accumulator + currentValue }) console.log(summedNums) // Output: 17

Here are the values of accumulator and currentValue as we iterate through the numbers array:

| **Iteration** | **accumulator** | **currentValue** | **return value** |
| --- | --- | --- | --- |
| First | 1 | 2 | 3 |
| Second | 3 | 4 | 7 |
| Third | 7 | 10 | 17 |

Now let's go over the use of .reduce() from the example above:

* numbers is an array that contains numbers.
* summedNums is a variable that stores the returned value of invoking .reduce() on numbers.
* numbers.reduce() calls the .reduce()method on the numbers array and takes in a callback function as argument.
* The callback function has two parameters, accumulator and currentValue. The value of accumulatorstarts off as the value of the first element in the array and the currentValue starts as the second element. To see the value of accumulator and currentValue change, review the chart above.
* As .reduce() iterates through the array, the return value of the callback function becomes the accumulator value for the next iteration, currentValue takes on the value of the current element in the looping process.

The .reduce() method can also take an optional second parameter to set an initial value for accumulator (remember, the first argument is the callback function!). For instance:

const numbers = [1, 2, 4, 10]; const summedNums = numbers.reduce((accumulator, currentValue) => { return accumulator + currentValue }, 100) // <- Second argument for .reduce() console.log(summedNums); // Output: 117

Here's an updated chart that accounts for the second argument of 100:

| **Iteration #** | **accumulator** | **currentValue** | **return value** |
| --- | --- | --- | --- |
| First | 100 | 1 | 101 |
| Second | 101 | 2 | 103 |
| Third | 103 | 4 | 107 |
| Fourth | 107 | 10 | 117 |

**Iterator Documentation**

There are many additional built-in array methods, a complete list of which is on the[MDN's Array iteration methods page](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array#Iteration_methods).

The documentation for each method contains several sections:

1. A short definition.
2. A block with the correct syntax for using the method.
3. A list of parameters the method accepts or requires.
4. The return value of the function.
5. An extended description.
6. Examples of the method's use.
7. Other additional information.

In the instructions below, there are some errors in the code. Use the documentation for a given method to determine the error or fill in a blank to make the code run correctly.

# Choose the Right Iterator

There are many iteration methods you can choose. In addition to learning the correct syntax for the use of iteration methods, it is also important to learn how to choose the correct method for different scenarios. The exercises below will give you the opportunity to do just that!

You'll see errors pop up in the terminal as you work through this exercise, but by the end the errors will be gone!

**Review**

Awesome job on clearing the iterators lesson! You have learned a number of useful methods in this lesson as well as how to use the JavaScript documentation from the Mozilla Developer Network to discover and understand additional methods. Let's review!

* .forEach() is used to execute the same code on every element in an array but does not change the array and returns undefined.
* .map() executes the same code on every element in an array and returns a new array with the updated elements.
* .filter() checks every element in an array to see if it meets certain criteria and returns a new array with the elements that return truthy for the criteria.
* .findIndex() returns the index of the first element of an array which satisfies a condition in the callback function. It returns -1 if none of the elements in the array satisfies the condition.
* .reduce() iterates through an array and takes the values of the elements and returns a single value.
* All iterator methods takes a callback function that can be pre-defined, or a function expression, or an arrow function.
* You can visit the [Mozilla Developer Network](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array) to learn more about iterator methods (and all other parts of JavaScript!).