**Introduction**

We are often unaware of the number of assumptions we make when we communicate with other people in our native languages. If we told you to “count to three,” we would expect you to say or think the numbers one, two and three. We assumed you would know to start with “one” and end with “three”. With programming, we’re faced with needing to be more explicit with our directions to the computer. Here’s how we might tell the computer to “count to three”:

for (let i = 1; i<=3; i++) {

console.log(i)

}

When we speak to other humans, we share a vocabulary that gives us quick ways to communicate complicated concepts. When we say “bake”, it calls to mind a familiar subroutine— preheating an oven, putting something into an oven for a set amount of time, and finally removing it. This allows us to *abstract* away a lot of the details and communicate key concepts more concisely. Instead of listing all those details, we can say, “We baked a cake,” and still impart all that meaning to you.

In programming, we can accomplish “abstraction” by writing functions. In addition to allowing us to reuse our code, functions help to make clear, readable programs. If you encountered countToThree() in a program, you might be able to quickly guess what the function did without having to stop and read the function’s body.

We’re also going to learn about another way to add a level of abstraction to our programming: *higher-order functions*. *Higher-order functions* are functions that accept other functions as arguments and/or return functions as output. This enables us to build abstractions on other abstractions, just like “We hosted a birthday party” is an abstraction that may build on the abstraction “We made a cake.”

In summary, using more abstraction in our code allows us to write more modular code which is easier to read and debug.

**Functions as Data**

JavaScript functions behave like any other data type in the language; we can assign functions to variables, and we can reassign them to new variables.

Below, we have an annoyingly long function name that hurts the readability of any code in which it’s used. Let’s pretend this function does important work and needs to be called repeatedly!

const announceThatIAmDoingImportantWork = () => {

console.log("I’m doing very important work!");

};

What if we wanted to rename this function without sacrificing the source code? We can re-assign the function to a variable with a suitably short name:

const busy = announceThatIAmDoingImportantWork;

busy(); // This function call barely takes any space!

busy is a variable that holds a *reference* to our original function. If we could look up the address in memory of busy and the address in memory of announceThatIAmDoingImportantWork they would point to the same place. Our new busy() function can be invoked with parentheses as if that was the name we originally gave our function.

Notice how we assign announceThatIAmDoingImportantWork without parentheses as the value to the busy variable. We want to assign the value of the function itself, not the value it returns when invoked.

In JavaScript, functions are *first class objects*. This means that, like other objects you’ve encountered, JavaScript functions can have properties and methods.

Since functions are a type of object, they have properties such as .length and .name and methods such as .toString(). You can see more about the methods and properties of functions [in the documentation](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function).

Functions are special because we can invoke them, but we can still treat them like any other type of data. Let’s get some practice doing that!

**Instructions**

**1.**

We’ve defined a function with a very long name: checkThatTwoPlusTwoEqualsFourAMillionTimes. This function takes a long time to execute. It checks whether 2 + 2 = 4, but it does it a million times (just to be really sure)! Create a shorter-named variable, is2p2 that will be easier to work with, and assign checkThatTwoPlusTwoEqualsFourAMillionTimes as its value.

Stuck? Get a hint

**2.**

Invoke your is2p2() function

Stuck? Get a hint

**3.**

Hmmm, if we forgot the original name of our function, is there a way we could figure it out? Use is2p2 to console.log() the name property of the function we assigned to is2p2. Check out [the documentation](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function) if you need a hint.

Hint

Access [the name property](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/name) of is2p2(). Just like with any object, you can use the dot operator to access a function’s property: functionName.name.

x

const checkThatTwoPlusTwoEqualsFourAMillionTimes = () => {

  for(let i = 1; i <= 1000000; i++) {

    if ( (2 + 2) != 4) {

      console.log('Something has gone very wrong :( ');

    }

  }

};

// Write your code below

const is2p2 = checkThatTwoPlusTwoEqualsFourAMillionTimes;

is2p2();

console.log(is2p2.name);

**Functions as Parameters**

Since functions can behave like any other type of data in JavaScript, it might not surprise you to learn that we can also pass functions (into other functions) as parameters. A *higher-order function* is a function that either accepts functions as parameters, returns a function, or both! We call the functions that get passed in as parameters and invoked *callback functions* because they get called during the execution of the higher-order function.

When we pass a function in as an argument to another function, we don’t invoke it. Invoking the function would evaluate to the return value of that function call. With callbacks, we pass in the function itself by typing the function name *without* the parentheses (that would evaluate to the result of calling the function):

const timeFuncRuntime = funcParameter => {

let t1 = Date.now();

funcParameter();

let t2 = Date.now();

return t2 - t1;

}

const addOneToOne = () => 1 + 1;

timeFuncRuntime(addOneToOne);

We wrote a higher-order function, timeFuncRuntime(). It takes in a function as an argument, saves a starting time, invokes the callback function, records the time after the function was called, and returns the time the function took to run by subtracting the starting time from the ending time.

This higher-order function could be used with any callback function which makes it a potentially powerful piece of code.

We then invoked timeFuncRuntime() first with the addOneToOne() function - note how we passed in addOneToOne and did not invoke it.

timeFuncRuntime(() => {

for (let i = 10; i>0; i--){

console.log(i);

}

});

In this example, we invoked timeFuncRuntime() with an anonymous function that counts backwards from 10. Anonymous functions can be arguments too!

Let’s get some practice using functions and writing higher-order functions.

**Instructions**

**1.**

Save a variable, time2p2. Assign as its value the result of invoking the timeFuncRuntime() function with the checkThatTwoPlusTwoEqualsFourAMillionTimes() function.

Stuck? Get a hint

**2.**

Write a higher-order function, checkConsistentOutput(). This function should have two parameters: a function and a value. It should call the argument function with the value two times. If the callback function produces the same result twice, it should return the result of the function call, otherwise, it should return the string 'This function returned inconsistent results'

Stuck? Get a hint

**3.**

Invoke your checkConsistentOutput() with the addTwo() function we wrote and a number as arguments.

Hint

Pass the function (not invoked) as an argument into your checkConsistentOutput() function.

const checkThatTwoPlusTwoEqualsFourAMillionTimes = () => {

  for(let i = 1; i <= 1000000; i++) {

    if ( (2 + 2) != 4) {

      console.log('Something has gone very wrong :( ');

    }

  }

};

const addTwo = num => num + 2;

const timeFuncRuntime = funcParameter => {

  let t1 = Date.now();

  funcParameter();

  let t2 = Date.now();

  return t2 - t1;

};

// Write your code below

const time2p2 = timeFuncRuntime(checkThatTwoPlusTwoEqualsFourAMillionTimes);

const checkConsistentOutput = (func, val) => {

    let firstTry = func(val);

    let secondTry = func(val);

    if (firstTry === secondTry) {

        return firstTry

    } else {

        return 'This function returned inconsistent results'

    }

};

checkConsistentOutput(addTwo, 10);

# Review

Great job! By thinking about functions as data and learning about higher-order functions, you’ve taken important steps in being able to write clean, modular code and take advantage of JavaScript’s flexibility.

Let’s review what we learned in this lesson:

* Abstraction allows us to write complicated code in a way that’s easy to reuse, debug, and understand for human readers
* We can work with functions the same way we would any other type of data including reassigning them to new variables
* JavaScript functions are first-class objects, so they have properties and methods like any object
* Functions can be passed into other functions as parameters
* A higher-order function is a function that either accepts functions as parameters, returns a function, or both

**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.

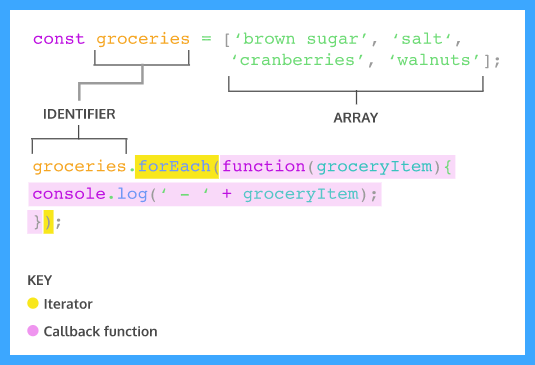
**Instructions**

Inspect the code in **main.js**. Notice the different methods being called on the arrays:

* .forEach()
* .map()
* .filter()

Run the code to see the output!

**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.  


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 forEach method 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.

**Instructions**

**1.**

Iterate over the fruits array to log I want to eat a plus the name of each fruit to the console. For example, I want to eat a mango.

You may use any form of callback you prefer.

Hint

Use dot notation to access the .forEach() method on the array and use a set of parentheses to call the method.

[1, 2, 3, 4].forEach()

.forEach() takes a callback function.

[1, 2, 3, 4].forEach(function(element){

console.log(`Printing ${element} to the console!`)

})

const fruits = ['mango', 'papaya', 'pineapple', 'apple'];

// Iterate over fruits below

fruits.forEach(a => console.log(`I want to eat a ${a}.`))

**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.

**Instructions**

**1.**

Add your code under the animals array and before the line console.log(secretMessage.join(''));

Use .map() to create a new array that contains the first character of each string in the animals array. Save the new array to a const variable named secretMessage.

Stuck? Get a hint

**2.**

Use .map() to divide all the numbers in bigNumbers by 100. Save the returned values to a variable declared with const called smallNumbers.

Hint

You may use any form of callback you prefer.

const animals = ['Hen', 'elephant', 'llama', 'leopard', 'ostrich', 'Whale', 'octopus', 'rabbit', 'lion', 'dog'];

// Create the secretMessage array below

const secretMessage = animals.map(ani => {return ani[0];});

console.log(secretMessage.join(''));

const bigNumbers = [100, 200, 300, 400, 500];

// Create the smallNumbers array below

const smallNumbers = bigNumbers.map(big => { return big/100;});

console.log(smallNumbers);

**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 true or false depending on the element that is passed to it. The elements that 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 6 characters 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.

**Instructions**

**1.**

Call the .filter() method on randomNumbers to return values that are less than 250. Save them to a new array called smallNumbers, declared with const.

Hint

To call .filter() on an array you can use the following syntax:

const numbers = [10, 20, 30, 40, 50];

const filteredNums = numbers.filter(function(number){

if (number < 25) {

return true;

}

})

Which can be refactored into:

const filteredNums = numbers.filter(function(number){

return number < 25;

})

You can also use arrow function syntax:

const filteredNums = numbers.filter(number => number < 25);

Pick one that works for you!

**2.**

Now let’s work with an array of strings. Invoke .filter() on the favoriteWords array to return elements that have more than 7 characters. Save the returned array to a const variable named longFavoriteWords.

Hint

To check the number of characters of a string you can use the .length property.

const randomNumbers = [375, 200, 3.14, 7, 13, 852];

// Call .filter() on randomNumbers below

const smallNumbers = randomNumbers.filter(num => {

  return num < 250;

})

const favoriteWords = ['nostalgia', 'hyperbole', 'fervent', 'esoteric', 'serene'];

// Call .filter() on favoriteWords below

const longFavoriteWords = favoriteWords.filter(word => {

  return word.length > 7;

})

**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 lessThanTen evaluates 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

**Instructions**

**1.**

Invoke .findIndex() on the animals array to find the index of the element that has the value 'elephant' and save the returned value to a const variable named foundAnimal.

Stuck? Get a hint

**2.**

Let’s see if we can find the index of the first animal that starts with the letter 's'.

Call .findIndex() on the animals array return the index of the first element that starts with 's'. Assign the returned value to a const variable named startsWithS.

Hint

To check the first character of a string you can use the access the 0th index:

const sampleString = 'treasure trove';

sampleString[0]; // 't'

You can also use .charAt():

sampleString.charAt(0); // 't'

const animals = ['hippo', 'tiger', 'lion', 'seal', 'cheetah', 'monkey', 'salamander', 'elephant'];

const foundAnimal = animals.findIndex(num => {return num === 'elephant'; });

const startsWithS = animals.findIndex(num => {return num[0] === 's'; });

**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 accumulator starts 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 |

**Instructions**

**1.**

Let’s practice calling .reduce() and using console.log() to check the values as .reduce() iterates through the array.

In **main.js**, there is an array of numbers, newNumbers.

To start, declare a new variable named newSum using the const keyword. Assign to newSum the value of calling .reduce() on newNumbers. You don’t need to provide any arguments to .reduce() yet.

You’ll also see a TypeError: undefined is not a function but we’ll fix that after we add our callback function in the next step!

Stuck? Get a hint

**2.**

Provide .reduce with an argument of a callback function. The callback function has two parameters. The first parameter is accumulator and the second parameter is currentValue. Use either a function expression or an arrow function.

Stuck? Get a hint

**3.**

To check the value being used as we iterate through the array, add three statements to the function body of the callback:

* console.log('The value of accumulator: ', accumulator);
* console.log('The value of currentValue: ', currentValue);
* a return statement that adds accumulator to currentValue.

Stuck? Get a hint

**4.**

Log the value of newSum to the console to see the return value of calling .reduce() on newNumbers.

**5.**

While we have this code set up, let’s also check what happens if you add a second argument to .reduce(). The second argument acts as an initial value for the accumulator.

Add a second argument of 10 to .reduce().

Hint

Make sure you’re adding a second argument to .reduce() and not another argument to the callback function.

Your code should look like:

const newSum = newNumbers.reduce(function(accumulator, currentValue){

// Existing code...

}, 10)

const newSum = newNumbers.reduce((accumulator, currentValue) => {

// Existing code...

}, 10)

/\*My Code -

const newNumbers = [1, 3, 5, 7];

const newSum = newNumbers.reduce((a,b) => {

console.log(`${a} - ${b}`);return a+b;})\*/

const newNumbers = [1, 3, 5, 7];

const newSum = newNumbers.reduce((accumulator, currentValue) => {

  console.log('The value of accumulator: ', accumulator);

  console.log('The value of currentValue: ', currentValue);

  return accumulator + currentValue;

}, 10);

console.log(newSum);

**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.

**Instructions**

**1.**

In the code editor, there is an array called words. We want to create a new array of interesting words. The first thing we want to do is check if there are words that are fewer than 6 characters long. There is something missing in the words.some() method call. Fix this method so that true is printed to the console.

For more information browse [MDN’s documentation on .some()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/some).

Stuck? Get a hint

**2.**

The .some() method returned true, which means there are some words in the array that are shorter than six characters. Use the .filter() method to save any words longer than 5 characters to a new variable called interestingWords, declared with const.

We’ve used .filter() in a previous exercise, but, for more information browse [MDN’s documentation on .filter()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter).

**3.**

In the last line of **main.js**, there is this code:

// console.log(interestingWords.every(word => ));

Complete the code between the parentheses to check if every element has more than 5 characters.

Make sure to uncomment (delete the // before) the last line of the program before you run the code.

For more information browse [MDN’s documentation on .every()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/every).

Hint

After the =>, return elements that have length greater than 5.

const words = ['unique', 'uncanny', 'pique', 'oxymoron', 'guise'];

// Something is missing in the method call below

console.log(words.some(word => {

  return word.length < 6;

}));

// Use filter to create a new array

const interestingWords = words.filter((word) => {return word.length > 5});

// Make sure to uncomment the code below and fix the incorrect code before running it

console.log(interestingWords.every((word) => {return word.length > 5}));

# 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!

**Instructions**

**1.**

Replace the word method in the first method call with a method that will do something to each of the values in the array and return undefined.

Stuck? Get a hint

**2.**

In the second method call, replace the word method with a method that will return a new array with only those elements longer than 7 characters.

**3.**

In the third method call, replace the word method with a method that take an array that contains multiple values and returns a single value.

Hint

This iteration method will take the elements of an array and return a single value.

**4.**

In the fourth method call, replace the word method with a method that will return a new array of numbers returned from the function.

Hint

What method would create new array that will have the same length of the original array?

**5.**

In the fifth method call, replace the word method with a method that will return a boolean value.

Hint

There are two iteration methods that you learned about in this lesson that would return a boolean value, true or false.

const cities = ['Orlando', 'Dubai', 'Edinburgh', 'Chennai', 'Accra', 'Denver', 'Eskisehir', 'Medellin', 'Yokohama'];

const nums = [1, 50, 75, 200, 350, 525, 1000];

//  Choose a method that will return undefined

cities.forEach(city => console.log('Have you visited ' + city + '?'));

// Choose a method that will return a new array

const longCities = cities.filter(city => city.length > 7);

// Choose a method that will return a single value

const word = cities.reduce((acc, currVal) => {

  return acc + currVal[0]

}, "C");

console.log(word)

// Choose a method that will return a new array

const smallerNums = nums.map(num => num - 5);

// Choose a method that will return a boolean value

nums.some(num => num < 0);

**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!).

**Instructions**

If you want to challenge yourself:

* Define a callback function before you use it in a iterator.
* Chain two iteration methods on the same array.
* Use the optional arguments in an iterator to include the index or the entire array. (Check out [MDN’s Array iteration methods page](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array#Iteration_methods) for more information)
* Use .reduce() to take a multi-layered array and return a single layer array from scratch.