Programming JavaScript

Santosh Kalwar, 25.01.2022

JavaScript Topics

- Arrays (level I and level II)
- Practice
- Objects
- Practice
- Arrow functions
- Practice
- EcmaScript - Basics

Arrays

Arrays in JavaScript allow you to store multiple elements in the same variable. You can store numbers, strings, booleans, arrays, objects & more. These can be mixed within the same array. Here are some examples:

```javascript
const users = []; // empty array
const grades = [10, 8, 13, 15]; // array of numbers
const attendees = ["Sam", "Alex"]; // array of strings
const values = [10, false, "John"]; // mixed
```

Name variables containing arrays in the plural as arrays can contain more than one item. This will prove to be especially useful once we need to iterate over an array.

.length property

You can get the number of elements in an array by using the `.length` property. For example:

```
[].length; // 0
```

```
const grades = [10, 8, 13, 15];
grades.length; // 4
```

Note that **.length** is a property (pre-computed value) and not a function. That's why you should not have `()` after it.

Get element by index

Similarly to strings, you can get an element from an array by using the square bracket syntax `[]` with the `index` starting from 0.

For example:

```
const users = ["Sam", "Alex", "Charley"];
users[1]; //"Alex"
```

Adding an element

You can add an element to an array by using the `.push()` method.

```
const numbers = [10, 8, 13, 15];
numbers.push(20); // returns 5 (the new length of the array)
console.log(numbers); // [10, 8, 13, 15, 20];
```

`Array.push()` returns the new length of the array.

As you can see, `numbers.push(20)` returns 5 which is the length of the array. This is often confusing for a lot of developers which is why it is highlighted here. `.push()` will add an item to the array but also return the new length of the array.

Array.push() on MDN

Arrays & const

Even though the variable `numbers` was defined with `const`, we were able to push new data into it. That's because `const` means you can only **assign the variable once** when it's defined. But it doesn't mean the variable is immutable. Its content can change.

What's the benefit of declaring it as a `const` you ask? The benefit is that once you define it as an array, it will always stay as an array which means you can safely call array methods on it. However, the array content can change.

```
const numbers = []; // start with empty array
numbers.push(10); // returns 1 (new length of array)
console.log(numbers); // [10] (still an array but content changed)
numbers.push(20); // returns 2 (new length of array)
console.log(numbers); // [10, 20] (still an array but content changed)
```

Recap

- `const data = [1, 2, 3]` is an array containing 3 numbers.
- `array.length` returns the number of elements inside the array.
- `array.push(x)` allows you to add the variable **x** to the array.
- `array.push(x)` returns the new length of the array (after the push has been made).
- Arrays defined with **const** are not constants because you can change the elements inside of it. However, you cannot re-assign them to another value thus they will always be an array.

Practice / Classroom coding

Go to your itsLearnign site, select Programming JavaScript

Select **Practice** folder, download the **Practice_lessons2308.zip** file

Open your code editor, e.g. VS Code / Atom / Sublime etc

If you get stuck, please check the "**hints**" folder for corresponding lessons

array01.js: Complete the function getEmptyArray such that it returns an empty array.

array02.js: Complete the function getNumberOfElements such that it returns the number of items contained in the elements array it receives.

array03.js: Complete the function useCalculator such that it adds the "Calculator" string to the apps array it receives.

array04.js: Complete the function such that the app variable is added to the apps array

array05.js: Complete the function getFirstApp such that it returns the **first** element from the apps array it receives as a parameter.

array06.js: Complete the function getLastApp such that it returns the **last** element from the apps array it receives as a parameter.

Array forEach

Array iteration is one of the most important concepts that you will use in JavaScript.

Let's say we have an array of grades and you'd like to loop (or iterate) over every item in this array. Here's how you do that in JavaScript:

```
const grades = [10, 8, 13];

grades.forEach(function(grade) {
    // do something with individual grade
    console.log(grade);
});
```

Always start with a `console.log()` inside your `.forEach` so that you can visualize the shift from array to array item.

The `.forEach(callback)` method allows you to run the `callback` function for every item in that array. Callbacks will be explained in more depth in the following chapter. For now, let's start with a basic definition.

Array forEach

A callback is a function definition passed as an argument to another function. In our example above, here's the callback function:

```
function(grade) {
    // do something with individual grade
    console.log(grade);
}
```

This callback function **receives** a `grade` and then logs it to the console. This is a function definition because it's not being executed. It only defines the behavior of the function. However, this function definition is passed as an argument to the `.forEach()` method:

```
grades.forEach(insert_callback_here);
```

Once you combine the two together, as in, pass the function definition as an argument to the `.forEach()` method, then you get:

Array forEach

```
grades.forEach(function(grade) {
    // do something with individual grade
    console.log(grade);
});
```

and this code will log **every** grade from the grades array to the console. So you will get:

```
10
8
13
```

So you can visualize the function calls as following:

```
// this is the callback
function(grade) {
    console.log(grade);
}
// call the callback with grade = 10
console.log(grade); // will log 10
// call the callback with grade = 8
console.log(grade); // will log 8
// call the callback with grade = 13
console.log(grade); // will log 13
```

Notice how the function definition is being called for every item in the array!

But who's calling it and providing the different values? Well, JavaScript is! You provide the callback (function definition) and pass it to the `.forEach()` and JavaScript does the rest!

How does it know that it’s “grade”

A common question when learning about callbacks is how does JavaScript know that grades becomes grade in the callback parameter. The answer is, it doesn't! JavaScript doesn't really care what you call your variables, it will **always** (in the case of .forEach) look for the **first parameter** you define in your callback function and pass to it the correct value.

So the code below does work (but please don't use it):

```
grades.forEach(function(x) {
    // this works, but avoid using generic variable names
    console.log(x);
});
```

So this works because JavaScript will look for the first parameter, x and it will call the callback and give a value to x every time.

Even though it works, you should always give clear variable names.

See **Array.forEach() on MDN**

Practice / Classroom coding

array07.js: Complete the function `loopThroughElements` such that it iterates over **every** item in the `elements` array it receives and logs it to the console (using `console.log`).

array08.js: Complete the function `logUserIds` such that it iterates over **every** item in the `userIds` array it receives and logs it to the console (using `console.log`).

Note: Feel free to check hints if you get stuck!

Return confusion

In this lesson, we'll cover a common confusion when it comes to returning inside a function that contains a `.forEach()` call.

But first, let's talk a bit more about naming variables when iterating.

Naming variables

Naming variables with a clear name makes it much easier for you and others to understand the code. This is especially true with iteration.

Thus, it's always a good idea to use the **plural** for the *array* and **singular** for the *item* of the array.
Here are some examples:

- **grades** => item is **grade**
- **people** => item is **person**

It may sound like it's a minor tip, but it makes a big difference! 💡

Return confusion

Here is a code example based on the above:

```javascript
const grades = [10, 14, 15]; // array (plural)
grades.forEach(function(grade) { // array item (singular)
    console.log(grade);
});
```

```javascript
const people = ["Sam", "Alex"]; // array (plural)
people.forEach(function(person) { // array item (singular)
    console.log(person);
});
```

Returning from loop

There's a common mistake that occurs when you try to `return` from a function that contains a forEach call. That's because there are 2 functions. Let's say you have this function:

```
function logUserIds(userIds) {
    userIds.forEach(function(userId) {
        console.log(userId);
    });
}
```

and you would like this function to `return true` when it's completed. Where do you place the `return true`?

Returning from loop

Would it be:

```
function logUserIds(userIds) {
    userIds.forEach(function(userId) {
        console.log(userId);
        return true; // does this work as expected?
    });
}
```

or:

```
function logUserIds(userIds) {
    userIds.forEach(function(userId) {
        console.log(userId);
    });
    return true; // or is this the correct way?
}
```

If we take a step back, the `return` keyword returns from **its own function**. Thus, the first approach does NOT work. Because you're returning from the callback function that the `.forEach()` receives. This will NOT return from the `logUserIds` function.

Returning from loop

Thus, the correct answer is the 2nd option:

```
function logUserIds(userIds) {
    userIds.forEach(function(userId) {
        console.log(userId);
    });
    return true; // ✅ return from the logUserIds function
}
```

The return true inside the function is not really useful because it's going to return from the callback function but there's isn't any more code inside that function anyway. The next iteration of the .forEach() will still happen.

Returning from loop

Let's take a look at an educational example to make sure you understand it. What do you think this function `logGrades` will return once its called?

```javascript
function logGrades(grades) {
    grades.forEach(function(grade) {
        console.log(grade);
        return 10;
    });
    return 20;
}
```

Will the function return 10 or 20?

The function will return 20 because it's returning from the outer function.

Note: You may have noticed that we're using the `function` keyword instead of modern JavaScript's arrow functions. This is purposely the case as functions are quite common everywhere. Arrow functions will be introduced soon and then used in array iteration.

Practice / Classroom coding

array09.js: Complete the function `sumGrades` such that it returns the sum of all the grades it receives as a parameter. We haven't seen **reduce** yet, so try to solve it using what you have learned so far.

array10.js: Complete the function `sumPositiveNumbers` such that it returns the sum of all positive numbers from the `numbers` parameter it receives.

array11.js: Complete the function `sumOddNumbers` such that it returns the sum of all the odd numbers from the `numbers` parameter it receives.

Note: Feel free to check hints if you get stuck!

Chapter recap

- `const data = [1, 2, 3]` is an array containing 3 numbers.
- `array.length` returns the number of elements inside the array.
- `array.push(x)` allows you to add the variable **x** to the array.
- `array.push(x)` returns the new length of the array (after the push has been made).
- Arrays defined with **const** are not constants because you can change the elements inside of it. However, you cannot re-assign them to another value thus they will always be an array.
- `.forEach(callback)` iterates over every item in an array.
- A callback is a function definition passed as an argument to another function.
- Always start with a `console.log()` inside the `.forEach()` to visualize the shift from array to array item (you can skip that when you become used to it).
- The `.forEach()` method will take your function definition and call it for every item of the array. Every time it calls it, it will replace the first parameter with the corresponding array item.
- Name your arrays in plural and the array item (inside the `.forEach()`) in singular.
- Make sure to correctly place the `return` inside a function that contains a `.forEach()`.

Array filter

We'll explore in this chapter more array methods while simultaneously learning about callbacks.

A common array **method** is the `.filter()` method. When you call this method on an array, you will get back another array that contains some of the items from the original array, based on the condition you specify. Let's take an example:

```
let numbers = [9, 5, 14, 3, 11];

let numbersAboveTen = numbers.filter(function(number) {
    return number >= 10;
});
console.log(numbersAboveTen); // [14, 11]
```

Don't forget the `return` keyword inside the callback function.

Notice how we got back a new array that contains the items which have satisfied the condition. The condition is that the `number` must be 10 or above.

In a couple of chapters, we'll see how to write the above with arrow functions, it'll look nice and short!

So how does this work?

Array.filter(callback)

Let's see how the above code works by breaking down its execution step by step.

The .filter() method expects a callback as the first argument. In our example, the callback is:

```
function(number) {
    return number >= 10;
}
```

JavaScript will take your callback and call it for **every single item** in the array. Our numbers array has 5 items, so it will call it 5 times. Every time that it calls this function, it will give a value to the number parameter that you specified inside this callback.

1. The first time it runs, it will give the number a value of 9 (the first item of the array).
2. The second time it runs, it will give the number a value of 5 (the second item of the array).
3. and so on and so forth until the last item of the array.

This is how callbacks work. Now every array method has a different behavior which I'll be explaining. This behavior often depends on the result of the callback. In this example, if the callback function returns true, then the item will be included in the final array returned by .filter(). However, if the callback function returns false, then the item will **not** be included in the final array.

Array.filter(callback)

That means, if you have the code below:

```
numbers.filter(function(number) {
    return true;
});
```

This will return **every** item in the array. So you will end up getting a copy of the original array. That's because the callback is always returning true. This code is not very useful, but it's to show you the importance of what the callback function returns and how that affects the result of the `.filter()` method.

This is why we had a condition `number >= 10`. This condition will return either `true` or `false` depending on the value of the `number`.

Array.filter(callback)

You probably have one question in your head now. How does JavaScript know that every item from the `numbers` array becomes `number` in the callback argument? And the answer to that is that it doesn't know!

JavaScript will take your callback and pass the item of the array as the **first parameter** to your callback function. This means that the code below works (but is not recommended):

```
let numbers = [9, 5, 14, 3, 11];

// works but is NOT recommended
let numbersAboveTen = numbers.filter(function(x) {
    return x >= 10;
});
console.log(numbersAboveTen); // [14, 11]
```

JavaScript doesn't care about what you call your variables. It will call your callback function and give a value to the first parameter which we called `x` here.

However, from a developer perspective what is `x`? It's not clear at all so always make sure to follow the **plural -> singular** naming convention that we covered in the previous chapter. It will make your life easier.

Array.filter(callback)

The code below:

```
let years = [2000, 2008, 2020, 2023];

years.filter(function(year) {
    return year >= 2010;
});
```

can be read in plain English as **Filter the years where the year is 2010 and above**.

- **Filter the years** corresponds to `years.filter`.
- **where the year is 2010 and above** corresponds to the callback function (and the condition inside of it) which is run for every item in the array.

Array.filter() on MDN

Chapter recap

- The `.filter()` method returns a new array that contains some of the items from the original array, based on the condition you specify.
- JavaScript will take your callback function and call it for every single item in the array.
- For the `.filter()` method, the result of the callback function matters. When it's `true`, the item will be included in the resulting array. Otherwise, it won't.
- JavaScript cannot make a smart guess that the `numbers` array becomes the `number` parameter in your callback function. What it does is that it calls your callback function while giving a value for the **first parameter** that you specified.
- Use the **plural -> singular** naming convention when using the `.filter()` method.

Practice / Classroom coding

array12.js: Complete the function `getPositiveTemperatures` such that it returns an array containing the positive temperatures (the temperatures that are above 0).

array13.js: Complete the function `getFreezingTemperatures` such that it returns an array containing the freezing temperatures (the temperatures that are below 0).

Note: Feel free to check hints if you get stuck!

Array find

We learned in the previous lesson about the Array .filter() method. In this lesson, we'll explore the `.find()` method which is a little bit similar in the way it works.

Let's start with an example, this time with an array of strings:

```
let names = ["Sam", "Alex", "Charlie"];

let result = names.find(function(name) {
  return name === "Alex";
});
console.log(result); // "Alex"
```

When you call the `.find(callback)` method on an array, you will get back **the first item** that matches the condition that you specify. If no items were found, you will get back `undefined`.

Array find

The condition that we specified here is that the name should be equal to "Alex".

So the .find(callback) method will call the callback that you provided for every item in the array until one of the callbacks returns true. When this happens, it will stop calling the remaining callbacks and return to you the item for which the callback returned true.

In our example above, here's the callback:

```
function(name) {
  return name === "Alex";
}
```

which gets called for name = "Sam" (first item of the array). However, the callback will return false because name === "Alex" returns false. So the callback will be called again with the next value of name. This time, name = "Alex". The callback will return true because name === "Alex" (the condition inside the callback) returns true. So the .find() method stops and returns to you that item which is "Alex".

Array find

Let's take another example but this time with an array of numbers:

```
let numbers = [9, 5, 14, 3, 11];

let firstNumberAboveTen = numbers.find(function(number) {
    return number > 10;
});
console.log(firstNumberAboveTen); // 14
```

Notice how even though there are 2 numbers that satisfy the condition, the `.find()` method stops at the **first** one that satisfies the condition.

This will bring us to the next section, which is `.filter()` vs `.find()`. What are the differences?

Array find

.filter() vs .find()

So, what is the difference between .filter() and .find()?

The difference has to do with the **return type** of these 2 methods:

1. The .filter() method **always** returns an array.
2. The .find() method returns the first array item that matches the callback function or undefined.

Let's take a look at a few examples:

```
let numbers = [9, 5, 14, 3, 11];

// filter() ALWAYS returns an array
numbers.filter(function(number) {
    return number >= 12;
}); // [14]

// .find() returns the first match or undefined
numbers.find(function(number) {
    return number >= 12;
}); // 14
```

Notice how the .filter() is returning an array, even if there's only 1 item that matches your condition. In contrast, the .find() method will return the first item that matches the condition.

.filter() always returns an array. Even if it matched one item or no items.

Array find

Now let's take a look at an example where no items satisfy the condition:

```
let numbers = [9, 5, 14, 3, 11];

// filter() ALWAYS returns an array (even if it's empty)
numbers.filter(function(number) {
    return number >= 15;
}); // []

// .find() returns the first match or undefined (when none of the items satisfy the condition)
numbers.find(function(number) {
    return number >= 15;
}); // undefined
```

Notice how the `.filter()` returned an empty array and the `.find()` returned `undefined`.

`.find(callback)` can return `undefined`. You may have to wrap its result in an `if condition` to avoid unexpected errors if you end up calling a method on its result.

Practice / Classroom coding

array14.js: Complete the function `getYear` such that it returns the `searchYear` (passed as 2nd parameter) when it's found in the array. Otherwise, it should return `undefined`.

array15.js: Complete the function `getOddYears` such that it returns **all** the years that are **odd** from the `years` parameter it receives.

Note: Feel free to check hints if you get stuck!

More array methods

In this lesson, we'll take a look at two array methods.

Array map

The `.map(callback)` method allows you to **transform** an array into another one. Here are some common examples:

- `[4, 2, 5, 8]` transformed to `[8, 4, 10, 16]`. We doubled every item in the original array.
- `["sam", "Alex"]` transformed to `["SAM", "ALEX"]`. We upper cased every item in the original array.

Notice that you always get back an array containing the **same number of items** compared to the original array, but every item has most likely undergone some transformation.

In the first example, the transformation is that we multiply every number by 2.
In the second example, the transformation is that we call `.toUpperCase()` on every item.

More array methods

Let's take a look at how we can implement these transformations:

```
const numbers = [4, 2, 5, 8];

const doubled = numbers.map(function(number) {
    return number * 2;
});
console.log(doubled); // [8, 4, 10, 16]
```

and

```
const names = ["sam", "Alex"];
names.map(function(name) {
    return name.toUpperCase();
});
```

If you forget the `return` inside the callback function, you will end up with the following array: `[undefined, undefined]`. That's because, for every item in the original array (`["sam", "Alex"]`), you're returning `undefined` so the end result will be `[undefined, undefined]`.

Once you make this mistake a few times, it becomes a clear signal that you've forgotten the return keyword.

Array includes(item)

The array `.includes(item)` method is one of the simplest array methods as it takes an `item` rather than a callback and returns `true` when that `item` exists in the array and `false` otherwise. Here's an example:

```
const groceries = ["Apple", "Peach", "Tomato"];
```

```
groceries.includes("Tomato"); // true
groceries.includes("Bread"); // false
```

Array join(glue)

When you have an array and you render this array to a web page (as we'll see later on in the DOM section of the course), the array will be automatically converted to a string. JavaScript will automatically invoke the .toString() method of the array which returns a string of the array elements separated by commas. Here's how it works:

```
const groceries = ["Apple", "Peach", "Tomato"];
groceries.toString(); // "Apple,Peach,Tomato"
```

But there's a downside, which is that you cannot customize the glue that gets inserted in between the array items, which is the comma , character.

If you'd like to customize the glue, then you can use the .join(glue) method:

```
const groceries = ["Apple", "Peach", "Tomato"];
groceries.join("; "); // "Apple; Peach; Tomato"
groceries.join(" . "); // "Apple . Peach . Tomato"
```

Some more arrary methods

We haven’t touched some more arrary methods from MDN site, let us check them now:

- **shift**	https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/shift
- **pop**	https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/pop
- **unshift**	https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/unshift
- **slice**	https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/slice
- **indexOf**	https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/indexOf
- **splice**	https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/splice
- **concat**	https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/concat

In this section, we will go to the MDN site and check above methods.

Practice / Classroom coding

array16.js: Complete the function `isAppUsed` such that it returns `true` when the `app` parameter it receives exists in the `apps` parameter, and `false` otherwise.

array17.js: Complete the function `getStringSizes` such that it returns an array of the number of characters for every string it receives in the `strings` parameter. This means, for the array `["abc", "d"]` it should return `[3, 1]` that's because the first string is made up of 3 characters and the second string is made up of 1 character.

Extra/classroom1.js: Complete the 10 short excercise related to arrow methods

Note: Feel free to check hints if you get stuck!

Objects

An object is a data type that allows you to group several variables together into one variable that contains keys and values. This is often used to represent or describe an entity. For example, a person, a user, a product, etc.

Here's how you can create an object:

```
const user = {
    id: 1,
    firstName: "Sam",
    lastName: "Doe",
    age: 20
};
```

It's recommended that you use camelCase for property names (for example `firstName` instead of `first_name`). See what is camelCase

https://en.wikipedia.org/wiki/Camel_case

Read the value of a property

To read the value of a property in an object, you can use the **dot notation** as following:

```
const user = {
    id: 1,
    firstName: "Sam",
    lastName: "Doe",
    age: 20
};
```

```
user.id; // 1
user.firstName; // "Sam"
user.isAdmin; // undefined (property does not exist)
```

It's a good idea to visualize the objects you're working with by logging them to the console.

Later in this course, we'll see how to read the value of a dynamic property.

Updating property value

You can also update a property value using the same **dot notation** followed by an equal sign:

```
const user = {
    id: 1,
    firstName: "Sam",
    lastName: "Doe",
    age: 20
};

user.lastName = "Blue";
user.age = user.age + 1;
console.log(user); // {id: 1, firstName: "Sam", lastName: "Blue", age: 21}
```

You are able to update the property value of an object defined by `const` because `const` does **not** mean that the variable is a constant, it just means that you cannot re-assign it. Thus, the variable is always an object, but its content (properties) can change.

Objects - recap

- An object is a data type that allows you to group several variables together into one variable that contains keys and values.
- It's recommended that you use camelCase for property names (for example `firstName` instead of `first_name`).
- To read or update the value of a property, you can use the **dot notation**.

Practice / Classroom coding

objects01.js: Complete the function `getProductDetails` such that it returns an object with the following properties:

- `id`: a number representing the id of the product.
- `title`: a string representing the title of the product.
- `inStock`: a boolean representing whether the product is in stock or not.

Give these properties any value you'd like, as long as it fits the expected return type.

objects02.js: Complete the function `getWeather` such that it returns the string:

`It's currently X degrees in Y`

Where `X` is replaced by the value and Y is replaced by the city name. These values are provided as a `city` parameter which is an object.

objects03.js: Complete the function `incrementAge` such that it returns the `person` object with the age incremented (add 1 to the existing value).

Note: Feel free to check hints if you get stuck!

Arrow func - Default parameters

Before we learn about arrow functions, let's take a look at a small but useful feature.

Say we've got the following code:

```
function addOne(number) {
    return number + 1;
}

addOne(2); // 3
addOne(5); // 6
addOne(); // what is returned?
```

What do you think will happen if you call `addOne()` (without any argument)? In some programming languages this would fail. But, in JavaScript, it won't fail and your code will continue executing.

The `number` parameter will receive a value of `undefined` so the function will return `undefined + 1` which results in `NaN`. We've seen in a previous chapter that we can prevent this by adding an `if` condition.
However, there's an easier way. Let's take a look at **default parameters**.

Note: if you're confused by the difference between parameters and arguments, here's a short definition: A parameter is a variable in a function definition. When a function is called, the arguments are the data you pass into the method's parameters.

Arrow func - Default parameters

Default parameters

Default parameters allow you to give a default value for one or more parameters that have not been provided when the function is called. Let's fix the example above and default the `number` to **0**:

```
function addOne(number = 0) {
    return number + 1;
}
```

```
addOne(2); // 3
addOne(5); // 6
addOne(); // 1
```

When you call the function `addOne()` without any arguments, it'll use the default value you defined in the function signature (`number = 0`) which is why we get back `1` from the function. See how we were able to prevent `NaN` without adding an `if` condition?
Also, notice how `addOne(2)` still works as before. So, when you do provide an argument, the default parameter is ignored.

Arrow func - Default parameters

Here's one more example:

```
function welcomeUser(name = "user") {
    return `Hello ${name}`;
}

welcomeUser("Sam"); // "Hello Sam"
welcomeUser(); // "Hello user"
```

Introduction to arrow functions

In this lesson, we'll get to know how to write arrow functions. An arrow function has 3 main benefits:

1. It's shorter to write.
2. It uses lexical scope (this will be explained in a later chapter as we need to learn classes first).
3. It can benefit from implicit return (covered later).

Here's an example of an arrow function:

```
const sum = (a, b) => {
    return a + b;
}
```

If you've never seen an arrow function, the syntax will most likely look weird at first. However, after you get used to it (might take a few days), you will notice that it's easy to read and easy to write.

From func. to arrow func.

In this section, we'll show you the steps that you need to go from a function to an arrow function. These steps are useful when introducing the concept of arrow functions. Once you get used to it, you will be able to write an arrow function directly.

There are multiple ways of writing a function in JavaScript.
You could either define a function and give it a name, or you could define a variable and assign it to an anonymous function.

So the following function:

```
function sum(a, b) {
    return a + b;
}
```

Can be written as:

```
const sum = function(a, b) {
    return a + b;
}
```

Notice how we define a variable `sum` and then we assign it to a function that takes 2 parameters `a` and `b`.

From func. to arrow func.

Now, let's convert that function into an arrow function.

You can do that in 2 steps:

1. remove the function keyword
2. add an arrow (`=` and `>`) between the parameters `(a, b)` and the opening curly brace `{`

This is how it will look like:

```
const sum = (a, b) => {
    return a + b;
}
```

Arrow functions always start with the parameters, followed by the arrow `=>` and then the function body.

Are func. deprecated?

No. You can keep writing normal functions, they are **not** deprecated. This is why the first we wrote some basic normal functions. You need to know how to write them because they are very common.
It is preferred that you use arrow functions though because of their benefits.

Practice / Classroom coding

arrowfunc01.js: You've been provided with a function `sum` that returns the sum of the parameters `a` and `b` it receives.However, in some cases, we're getting `NaN`. Fix this function without using `if` conditions (or ternary) so that we don't get `NaN`. Check the sample usage to see the expected result.

arrowfunc02.js: Our `logUserIds` function fails when we call it without any parameter. Fix that without using an **if** condition.

arrowfunc03.js: You are given a function `triple` that returns the result of its parameter multiplied by 3. Rewrite it as an arrow function.

arrowfunc04.js: In this practice, you're asked to write the `triple` function from scratch as an arrow function.The function `triple` receives a parameter and returns the result of multiplying it by 3. Feel free to add a sample usage at the end once you've written the `triple` function.

arrowfunc05.js: In this practice, you're asked to write the `sum` function from scratch as an arrow function. The function `sum` receives 2 parameters and returns the result of their sum. Feel free to add a sample usage at the end once you've written the `sum` function.

Extra/classroom2.js: In this practice, we will go through the several tasks which will try to cover what we have studied in Arrary methods chapter (check the latest practice zip file from itsLearning — Practice folder)

EcmaScript - Basic Concept

We know that JavaScript is the language that runs in the browser, it's the programming language you've been learning throughout this course. But what is EcmaScript?

JavaScript is an **implementation** of *EcmaScript*. It's actually the most popular implementation of EcmaScript.

EcmaScript is a specification for a scripting language managed by an organization called Ecma (Ecma International) which is headquartered in Geneva, Switzerland.

So, Ecma International defines all the rules and specifications for a scripting language called EcmaScript but these are just the specifications. You need an actual programming language to implement that, which is **JavaScript**.

EcmaScript vs JavaScript

EcmaScript is the specification, and JavaScript is the language.

When there's a new version of EcmaScript, this also means that there's a new version of JavaScript because JavaScript is the language implementing EcmaScript.

Vanilla JavaScript

You may encounter the popular term **Vanilla JavaScript**.

Vanilla JavaScript means JavaScript without a framework. So, pretty much everything we are learning here…

People often use the term Vanilla JavaScript when asking questions on the Internet as a way to communicate that they are looking for an answer purely in JavaScript without jQuery, or React, etc.

People often use the term Vanilla JavaScript when asking questions on the Internet as a way to communicate that they are looking for an answer purely in JavaScript without jQuery, or React, etc.

EcmaScript versions

Before the year 2015, developers were writing EcmaScript 5 (ES5). It's a specific version of JavaScript which was released in 2009 and became supported on most browsers.

Then in 2015, nearly 6 years later, a new version of JavaScript was released which was EcmaScript 6 (or ES6).

It was also decided that every year there would be a new version of JavaScript, which is why ES6 was renamed to ES2015 where the year would be visible in the version name.

So ES2015 is published in 2015, ES2016 is published in 2016, and so on and so forth.

Versions Recap

As a recap regarding the versions, ES5 (EcmaScript version 5) is unofficially referred to as the "old version of JavaScript". For example when you search for some code on StackOverflow and you find the `var` keyword. That is from ES5.

The "new version of JavaScript" is from ES2015 and beyond.

ES2015 was big update

ES2015 was a big update to the language. It brought most of the features we will touch upon in this course such as:

- let/const
- spread operator
- destructuring
- arrow functions
- lexical this
- and so much more…

Check out for example:

- https://babeljs.io/docs/en/learn/#ecmascript-2015-features

Adding features

Keep in mind that JavaScript is mostly receiving new features with every EcmaScript update. Even though it's a new "version", you won't have to learn it again from scratch. It only means that the language received new superpowers.

That's due to the nature of the web. Most websites are supposed to continue working even if they were written in 1990.

In some rare scenarios, some features are removed/modified from the language due to security flaws or similar concerns. But, the general rule is that new features get added, and if you keep on using the old ones they will still work.

Chapter Recap

- Before the year 2015, developers were writing EcmaScript 5 (ES5)
- ES2015 was a big update to the language. It brought most of the features you learned in this course
- As of ES2016, since the releases became yearly, there will only be a dozen of features per release.
- With every new EcmaScript version, JavaScript gains additional features.