# WELCOME TO REACT WEEK 1

Welcome back everyone! In this course, you will be learning to use the React.js front-end library, which means you will be writing a lot of JavaScript. This first week, you will focus on learning more JavaScript in order to prepare you to learn React.

Slides/Screenshots:

You will be learning about JavaScript objects, classes, advanced array methods, and more.

And you will spend a lot of this week in hands-on practice of those concepts, through multiple Code Challenges and a Quiz. At the end of this week, and every week, you will meet with your instructor and classmates to review what you have learned and work on a Workshop Assignment.

After that, your next four weeks will be spent learning React. In Week 2, expect to learn how to set up a React project and begin writing your first React components. You will work with the same “Nucamp” campsite review website you built in Bootstrap, but you will start from the ground up and build the same website using React.

In Week 3, you will learn about different types of React components, and about the React Router library to help you turn your React project into what’s called a Single Page Application, or SPA.

In Week 4, you will work with user input via forms, and you’ll learn about a state management library called Redux.

In Week 5, you will get even deeper into Redux, and you’ll learn about using the Fetch API and JavaScript promises to request data from a web server.

In addition to all this, starting in Week 2, you will take your Portfolio Project from your Bootstrap course and recreate it using React.

Expect to learn a lot during this course, and remember as always to stay focused, …, and don’t forget the 20 minute rule!

# 

# OBJECTS

*NOTES TO SELF: At some point discuss how const object properties and methods can be changed, const can be reassigned but it is not immutable.*

*To include:*

* *What is an Object, compare to Arrays*
* *How do you create an Object (object literal not new Object)*
* *What are properties and methods*
* *How do you access properties and methods of an object*
* *Talk about arrays of objects*
* *Talk about using array methods on object arrays, how to access properties of objects in arrays*
* *Talk about the exception with const and objects/classes (can change properties and methods as long as object itself is not reassigned, can change array item values as long as array is not reassigned)*

It’s finally time to talk about objects in JavaScript.

Remember that JavaScript has 8 data types:

Slide: [String Number .. etc]

In this lesson, you will learn about the Object data type. All of the other seven data types are classified as ‘primitive’ data types. Object is the only non-primitive data type in JavaScript - and while it’s grouped with data types, what it is really is more of a data structure. Rather than a single piece of data, like a number or a string, it’s a collection of data.

We can create an object and put it into a variable with a name, just like any other variable. Then here’s where it gets interesting -- we can create variables and functions that belong to that object, and these special variables and functions are called properties and methods, and they can only be used by referring to that object.

That’s not going to make a lot of sense without an example, so let’s go to our example.

Please open your developer console and type along with me.

We’ll look at an example of an object with just properties first. Let’s say we’re writing a game, and we have a monster in the game, let’s say a dragon.

[Slide: Creating Objects with Properties and Methods]

So we’re going to make an object for the dragon. It’s going to start out just like we’re making a variable declaration:

const dragon1 = { color: "red", maxHP: 1000 };

But then we’re going to use curly braces to surround what’s called the object literal, which is how we create the object.

Similar to how when you create an array, you use square brackets to surround the array items, and that’s actually called an array literal.

Don’t press enter yet.

Now, when we write objects, it’s common to put everything on separate lines like this:

const dragon1 = {

color: 'red',

maxHP: 1000

};

This kind of looks like a code block, like an if block or function block, but it is not. Each line does not contain a statement, and they’re not ended with a semicolon. We are defining the properties that belong to this dragon by using a property name, then a colon, then the property value

Then we’re separating each of these property definitions with a comma.

Outside of an object, when you create a variable, you use an equals sign as the assignment operator. When you create an object property, you use a colon. Remember that, it’s a common mistake to forget that and use the equals sign here.

So now that we have created this dragon, we can type the variable name we used for the object into the console and the console will show us what’s inside it.

We can query the properties that belong to it by using dot notation, and the way we can do that is by typing the name of the object, then a dot, then the property name. [type dragon1.color, dragon1.maxHP]

There’s a second way you can get a property, which is by using bracket notation instead of dot notation, but typically it’s longer to type and dot notation is preferrable. There are cases when bracket notation is useful, but we won’t discuss that right now. [type dragon1[‘color’]]

And notice, if you just try to get a property without first giving it the object name, it won’t work: [type maxHP]. The property is attached to the object. That’s what makes it a property and not a variable.

I hope you can already see how useful an Object can be. You can bundle a lot of related data together into a single Object.

# ARRAYS VS OBJECTS

You learned before that Arrays are a special type of Object. Arrays are not as flexible as objects. They can have items, just as objects can have properties, but the array items are always accessed by a numeric index, like this:

Array boxes 0 1 2 3

Whereas for Objects, the properties are accessed by the property names, which are configured by you to whatever you want them to be. Another way that objects are more flexible is that you can add custom methods to objects.

Remember, arrays have methods like array.join(), array.pop(). These are pre-defined methods in the javaScript language that are already written for you, and you can use them on any array.

However, with objects, you can add your own custom method. For example, let’s say that every time I wrote dragon.roar() I wanted to see the message “The dragon roars ferociously.”

Here’s how we would do this. Let’s make a new dragon. This one’s going to be a blue dragon.

const dragon2 = {

color: "blue",

maxHP: 1000,

roar() {

console.log(`The dragon roars ferociously.`);

}

};

You see that I used a method name, followed by a parameter list, which in this case is empty, then curly braces, and the method body inside the curly braces.

And now I can call it just like this: (later in recording, after “which brings us to our next topic”

Now, this is a shorthand syntax for creating a method that was added to JavaScript in ES6. Before ES6, you would have had to write it with the anonymous function expression syntax, like this:

const dragon3 = {

color: "black",

maxHP: 1000,

roar: function() {

console.log(`The dragon roars ferociously.`);

}

};

And I can call this method in the same way. (recording after next para)

These two ways are equivalent, and while I encourage you to use the shorthand, you will still run into plenty of code that uses this old way, so it’s worth knowing about it. Also, you will need to know it in order to add or update a method of an existing object. Which brings us to our next topic - how to add and update properties and methods to an existing object.

[SLIDE: ADD AND UPDATE PROPERTIES AND METHODS TO AN OBJECT]

Let’s say we want to add an “element” property to our dragon1 object. Let’s say we want this dragon to be a fire dragon, so we can write: dragon1.element = “fire”

And here, notice we did use an equals sign, the assignment operator, and not a colon, because we only use the colon when we’re writing the object literal inside curly braces. Outside of that, we must use the assignment operator.

So, if you want to add a property to an existing object, just use the object variable name, then add the property name that you want to use and assign a value to it just like any other variable.

And you can update the existing properties in the same way. Let’s update the “color” property:

dragon1.color = “crimson”;

And now if we look at our dragon1 object, we can see that the color has changed.

Next let’s look at adding or updating a method for an object. To add or update an already existing method, you will use the function expression syntax, like this:

dragon1.attack = function() { console.log("The dragon breathes a stream of white-hot flame at you!"); };

Let’s test that out and have our dragon1 attack: dragon1.attack();

And if we type the object name into the console, we can see that the method is inside it: dragon1

And if we want to update an existing method, we can just overwrite it like this:

dragon2.roar = function() { console.log(“The dragon lets out a tremendous roar!”); };

Okay, let’s stop for a minute. I know what you’re thinking. This doesn’t make any sense. These are constants! You thought we could never change constants after declaring them the first time. How are we changing these properties and methods?

For now, what you need to know is that even if an object variable is declared using the constant keyword, the properties and methods can indeed be changed, but the variable name can never be reassigned.

For example, try this and see what happens:

dragon1 = { color: “blue”, element: “water”, maxHP: 1000 };

Right, can’t do it. You cannot reassign the entire object. However, it’s totally fine to update the properties and values.

That’s actually true of arrays as well.

If I have a constant array like this: const colors = [“red”, “blue”, “black];

I cannot reassign the whole array: colors = [“crimson”, “turquoise”, “midnight”]

However, it’s completely fine to change an item in the array, like this:. Colors[1] = “crimson”

Colors[2] = “turquoise”

OK, I just have one more topic I want to cover in this lesson. Let’s look at using parameters in methods. You can use parameters in methods just like you do in functions.

Let’s revisit dragon2. Just for practice adding properties, we’ll give it an element property as well and make it a water dragon.

Now let’s give it the ability to swim, and let’s make it so that we can pass in the direction it swims to:

dragon2.swim = function(dir) { console.log(`The dragon swims away to the ${dir}.`); }

Now you can call this method like this, passing an argument into the parameter list:

dragon2.swim("north");

And you can see how that works.

We’ll stop here. You’ve learned what objects are, the meaning of object properties and methods, how to create and update objects and their properties and methods, and how to access those properties and call those methods. And you got a bonus lesson on how const works with objects and arrays. Objects are a complex topic, and of course there’s a lot more to objects, but now you have begun to get to know them.

Note that objects are a common data structure in programming, and the concept of an object is not unique to JavaScript. It’s part of an entire programming paradigm known as object-oriented programming.

Instead, you’re going to define properties and methods for the object.

you have a collection of key-value pairs where not only the values, but also the keys are defined by you. And instead of using numbers for the keys, you’re going to use words.

Here’s an example. Let’s write this together in the console:

myMealsToday = { breakfast: "pancakes", lunch: "tacos", dinner: "pizza" }

let myMealsToday= {  
 breakfast: “pancakes”,  
 lunch: “tacos”,

dinner: “pizza”

}

With an Object, the way you will typically access its values is by using the key like this:

The object name, dot, then the name of the key you want to retrieve the value for:

myMealsToday.breakfast

myMealsToday.lunch

myMealsToday.dinner

Remember before, we said that methods are functions that are tied to objects.

So the way to think about these is

I spoke before about arrays being a special kind of object. Arrays are a collection of values as well, with a fixed index that looks like this: 0 1 2 3 4. (Illustrate it with boxes)

With an Object, you no longer have a fixed index.

Let’s create an object called boat.  
const canoe = {  
 material: “wooden”,  
 lengthInFeet: 12,  
 paddle() {  
 console.log(“You paddle the canoe.”);  
 }  
}  
As you can see, we create an object using a variable declaration keyword. In this case I chose to use const. I gave the object a variable name of ‘boat’, then after the assignment operator, I used what’s called object literal syntax to create the object - that means I used a pair of curly braces, and inside it, I defined the key-value pairs for my object, each pair separated by a comma.

And the way you want to think about this is, an Object can have variables and functions that are specifically tied to that object. They belong to that object. The variables that belong to the object are called properties. You put the variable name on the left side, then a colon, and the value of that variable on the right side. You use a colon here for assignment instead of the equals symbol, remember that! That’s a common beginner’s mistake, so watch out for it.

The functions that belong to the object are called methods. Methods are functions that are tied to a specific object, OK? You can define methods for an object like this - just add the method name, followed by the argument list, even if it’s empty like this one, then .

So then you can access the properties and methods of this object by using the variable name you gave the object, then dot, then the property or method name. Since methods are functions, you need to make sure to call them by including the parameter list, even if it’s empty, just like witih functions:

Canoe.material  
canoe.paddle()

You could add parameters to this method just like with functions. I’ll refresh here and rewrite this to show you:

Const canoe = {   
 ..  
 paddle(dir) {  
 console.log(`You paddle the canoe ${dir}`);  
 }  
}

And now I can call it like this: canoe.paddle(“north”);

# 

# WHAT IS THIS

**Notes to self:**

* *What is ‘this’*
* *What is this inside an object*
* *What is the meaning of lexical scope*
* *What is this in relation to an arrow function*
* *What is this in relation to any other function*

I will now introduce you to using the *this* keyword in JavaScript. There’s a lot of complexity to the *this* keyword, and I’m not going to throw too much information at you right now. Instead, we’re just going to go over some examples together that will help you understand what *this* is and why we even care about it.

So go ahead and open up your developer console. And our dragon will be making a comeback.

const dragon1 = {

color: 'red',  
 roar() {  
 console.log(“The dragon lets out a tremendous roar!”);

};

So lets bring back our red dragon

and give it a roar method again

And you can see that roar method does this: [show]

But let’s say that we wanted to use the color of the dragon in the roar() method, but we didn’t want to hard code the color value, because it could change later.

What do you think? Maybe you can do this - let's try to update the method using the function expression syntax, as I showed you before,

we will change the quotes to backticks so that we can use the variable *color* inside the string with the template literal syntax, as you’ve learned about. Again, the backticks and the dollar symbol and curly braces together is the template literal syntax that makes it so we can interpolate a JavaScript variable inside of a string.

dragon1.roar = function() {

console.log(`The ${color} dragon lets out a tremendous roar!`);

}

well Let’s try that:

That does not work. Why not?

You see, the property color doesn’t exist on its own. To use it inside an object method, it needs to refer to an object, even though the method is inside an object.

So maybe we can do something like this:

dragon1.roar() = function() {

console.log(`The ${dragon1.color} dragon lets out a tremendous roar!`);

}

Let’s run that now… dragon1.roar()

Oh hey, that works.

But what if we wanted to be able to use this exact same method, no matter what object it’s on, without changing it at all?

Let’s bring back the blue dragon, and let’s give him the exact same method:

const dragon2 = {

color: 'blue',

roar() {

console.log(`The ${dragon1.color} dragon lets out a tremendous roar!`);

}

}

dragon2.roar()

Well obviously that’s not correct. So far the only solution would seem to be that you change the method for each object, like this, so we’re referring to dragon2.color in this one, and dragon1.color in the other one:

dragon2.roar() = function() {

console.log(`The ${dragon2.color} dragon lets out a tremendous roar!`);

}

That works, but wouldn’t it be nice if the roar method automatically knew what object it belonged to?

Enter this.

Here’s what will solve it:

dragon1.roar = function() { console.log(`The ${this.color} dragon lets out a tremendous roar!`); }

dragon2.roar = function() { console.log(`The ${this.color} dragon lets out a tremendous roar!`); }

dragon1.roar()

dragon2.roar()

The this keyword tells the method to look at the object that owns it for the color variable, and uses that. So you don’t have to go in and tell it, dragon1, dragon2, dragon3, which would be very annoying if you had an army of dragons.

This is just an introduction to the concept of the *this* keyword, and we’ll build on this knowledge as you continue with the course.

# CLASSES

You may be used to the word class in a CSS context, where you put classes on HTML elements then make rules for them in stylesheets. Let me make this clear right now - a class in JavaScript is something completely different. We are not talking about CSS classes nor the HTML class attribute.

Classes in JavaScript are templates for objects. They help you create multiple instances of the same kind of object. The concept of a class is common in object-oriented programming, so you will encounter it in other programming languages, not just JavaScript.

Let’s stick with our dragons example. Before, we had to create a new object from scratch every time we wanted to make a different dragon. Let’s look at how we can make dragons more easily using classes.

Go ahead and open your console and type this in with me. I’ll remind you that you won’t learn just by reading or watching, you really need to type this in yourself, in your own console.

OK, let’s make a dragon class.

[SLIDE: CREATING A CLASS,  
THE CLASS CONSTRUCTOR AND CLASS PROPERTIES  
CREATING INSTANCES WITH THE NEW KEYWORD  
class Dragon {

constructor(color, maxHP) {  
 this.color = color;  
 this.maxHP = maxHP;  
 }  
}

(narrate while typing): we start with the class keyword, then the name of the class, which must be capitalized. Then we will start a curly brace, and go to the next line.

The next thing we will do inside the class is write a constructor. The constructor is a special class method that will say what we want the default properties of our class to be - in this case, we want the default properties to be color, and maxHP. There can only be one constructor for a class, and you can leave it out if you don’t need your class to have any default properties.

Then inside the constructor, what we’re going to do is take these arguments that are passed into the parameter list and set them as the properties of the objects we’re going to make from this class. And we’re doing this just like a regular variable assignment, with the equals sign, and semicolons after each one.

Now we have our Dragon class. This is a template from which we can make as many dragons as we want. And to do this, we will use the new keyword. Let’s make our dragon1 again, using this class:

Const dragon1 = new Dragon(“red”, 1200);

And let’s make dragon2 again.

Const dragon2 = new Dragon(“blue”, 1 00);

Each time we make an object using a class like this, this object is called an instance of the Dragon class.

To step through it: this new keyword creates a new object, and it passes these argument values into the class constructor. and There they become properties of the new object through assigning them as values of the this, dot, propertyname.

Next we’ll look at Class Methods.

**[Slide: Class Methods]**

We can also add default methods to classes, but we won’t use the constructor for that. Instead, the way you add them will look pretty much identical to the way you add them to a single object. (not recorded: only difference is that you will not separate multiple methods with a comma) Let’s refresh the console and remake our Dragon class.

class Dragon {  
 constructor(color, maxHP) {  
 this.color = color;  
 this.maxHP = maxHP;  
 }  
 roar() {  
 console.log(`The ${this.color} dragon lets out a tremendous roar!`}  
 }  
}

I’m a little tired of typing so I’ll just use my up arrow key to bring up the dragon class definition I made before, and I’ll go in here and add a method. If I just hit enter here, though, I get this Uncaught SyntaxError: Unexpected end of input, message, so let me show you how to avoiid that.

Hit the up arrow, go down to where you want to hit enter, then use Shift-Enter instead. When you use Shift Enter, that will prevent the lines you type from being sent to the console just yet. Then wait to hit Enter until you're all finished typing in your code.

(while typing) We’ll add methods below the constructor.

const dragon1 = new Dragon("red", 1200);

const dragon2 = new Dragon("blue", 1000);

Then we can make our red and blue dragons again…

And now notice that if we use roar() on either object, we get the correct color.

You can see that the *this* keyword can be especially useful when writing class definitions. You can refer to the properties of the object to be created by the class, without needing to have the object name.

Because this week is meant to get you prepared for React, it’s important that you understand how classes work in JavaScript. However, I’m going to warn you in advance that when you finally do start learning React, React has its own particular way of working with classes called class components. You’ll see that React class components are based on JavaScript classes and will look quite similar, but will also have their own particular syntax. You’ll learn more about that next week, but I just wanted to give you a heads up about it in advance.

In this lesson, you’ve learned about how to use JavaScript classes with class constructors and methods to create JavaScript objects with the new keyword. The best way to learn is by doing, so go on to the Classes Code Challenge to practice using classes on your own.

done

class Cat{

constructor(name, color) {

this.name = name;

this.color = color;

}

meow() {

console.log(`The ${this.color} cat named ${this.name} meows.`);

}

}

const cat1 = new Cat("Ray", "gray");

TO MENTION:??

* Class declarations are not hoisted.
* JavaScript inside classes is in strict mode.
* How ‘this’ works inside classes.

# CLASS INHERITANCE

* What is class inheritance
* Give an example
* What is super() in the constructor

The concept of class inheritance is that we can take a class and make child classes from it. The child class then has all the default properties and methods of the parent class, but it can add more properties and methods without affecting the parent class.

Let’s take a look at what that means in the developer console:

(while typing)

Make sure you refresh the browser tab your console is in to clear out any variables.  
Typically, parent classes will be more general, and children classes will be more specific. So maybe instead of a Dragon class, we have a Monster class, and we’ll just give the Monster class a generic method for now:

class Monster {

roar() {

console.log(`The monster lets out a tremendous roar!`)

}

}

Then we’ll create a child class of Monster named Dragon, using the extends keyword. This extends keyword is the way you make a child class.   
And not all monsters can fly, but let’s say that all dragons can fly.

class Dragon extends Monster {  
 fly() { console.log("The dragon flaps its wings and begins to fly.";)

}

We’ll create a dragon from the Dragon class:

const dragon1 = new Dragon();

And now you can see that it has inherited the roar() method from the Monster class:  
dragon1.roar()

But it also has the fly method from the Dragon class:  
dragon1.fly()

If you created a monster from just the Monster class:  
The monster cannot fly because the Monster class doesn’t have that method.

So child classes inherit the methods of their parent classes, but not vice versa.

Next, we’ll talk about properties of parent and child classes, and how to use the **super** method in child classes.

**[SLIDE: SUPER AND PROPERTIES]**

Let’s refresh to clear the console and recreate our Monster and Dragon classes.

This time, we’ll give the Monster class a constructor for the type and color. (re-recorded down a few lines)

Remember, for any class to store default properties, it must have a constructor method.

We’ll also set a property that all Monsters have by default with the same value. We’ll say, isScary = true. And since all monsters have this, and it’s always true, we don’t have to pass in an argument, we’ll just set it right in the constructor.

Then we can use this.color and this.type inside the roar() method.

class Monster {

constructor(type, color) {

this.type = type;

this.color = color;

this.isScary = true;

}

roar() {

console.log(`The ${this.color} ${this.type} lets out a tremendous roar!`);

}

}

And this time when we make our Dragon child class using extends, we’ll also make a constructor, and we'll have the parameters of type, color and element passed in…

And we’ll use this super method, which is a special method we can use in child classes to, to put it simply, be able to inherit properties from the parent class. So we want to be able to use the type and color that are in the Monster class, so we’re going to put that in our super method.

class Dragon extends Monster {

constructor(type, color, element) {

super(type, color);

this.element = element;

}

fly() {

console.log(`The ${this.color} ${this.element} ${this.type} flaps its wings and begins to fly.`);

}

}

And even though we’re getting type and color in as arguments through the constructor parameter list here in the child class, we don’t need to say this.type=type or this.color = color, because we put it in our super method, so that happens up in the parent class.

But since the parent class doesn't have the element property, we will need to add this.element = element in the child class's constructor. And we’ll add the fly method and let's go ahead and use all three properties, color, element, type.

const dragon1 = new Dragon(“dragon”, "blue", "water");  
dragon1.roar()  
dragon1.isScary

And you can see that the dragon1 object created from the child class, Dragon, has received the roar method and the isScary property from the parent class, Monster.

dragon1.fly()

and it has its own fly method and element property, and the type and color properties have been created for this object properly.

Let’s trace through this together. When we created the new dragon, dragon1, we passed in these strings of “dragon” “blue” and “water”, which went into the constructor for the Dragon class.

Then here, this super method connected this child class's constructor with the constructor for the parent class Monster, and the type and color were, by using the this keyword here, assigned as properties of the new dragon1 object.

TO RECORD: The property isScary gets set to the object here in the Monster class constructor as well.

And the element variable was also made into a property of the dragon1 object, but that happened here in the Dragon class’s constructor.

Let's make another child class from the monster parent class, let’s say a werewolf:

class Werewolf extends Monster {

constructor(type, color) {

super(type, color);

}

howl() { console.log(`The ${this.type} howls loudly.`); }

}

And it has its own howl() method.

const werewolf1 = new Werewolf("werewolf", "gray");

Then just like the dragon, the Werewolf class also inherits the roar() method from the Monster class, and it has the isScary property, set to true.

But it can howl, which the dragon can't, and the generic monster can't either.

So you can see how useful class inheritance can be. When you’re going to have multiple classes that are going to share some of the same properties and methods, then you can set up parent and child classes so that you don’t have to write the same code over and over again.

# 

# DEFAULT FUNCTION PARAMETERS

The ability to set Default function parameters is a useful addition in ES6 JavaScript.

In JavaScript, it is possible to call a function without providing arguments for all the parameters.

For example, I could have a function like this:

function logSum(a, b) {

console.log(a + b);

}

This function takes two arguments and console logs their sum.

And normally, I would call this function like this:

logSum(2, 3);

But what if I called the function like this, giving it only one argument?:

logSum(2);

Well, JavaScript allows me to do that. However, any parameter for which I don't pass in an argument is by default set to undefined.

So what happened here was that this function tried to add 2 + undefined, which resulted in NaN.

One way to prevent this from happening would be to set default values inside the function, like this:

function logSum(a, b) {

if (a === undefined) { a = 0; }

if (b === undefined) { b = 0; }

console.log(a + b);  
}

Here, I'm checking to see if a or b are undefined, and if so, then I set the value to 0.

That way, I can still do this:

logSum(2)

And get back a number instead of NaN.

With default function parameters in ES6, we can actually set the default value for an parameter in the parameter list, like this:

function logSum(a = 0, b = 0) {

console.log(a+b);

}

This says that if no argument is passed into this parameter, then by default, initialize it to 0, instead of setting it to undefined.

When you are defining a class, you can use default function parameters in the class constructor. Here's an example:

class Cat{

constructor(name, color = "gray") {

this.name = name;

this.color = color;

}

meow() {

console.log(`The ${this.color} cat named ${this.name} meows.`);

}

}

const ray = new Cat("Ray");

ray.meow();

Since I didn't pass in a color, the color defaulted to "gray".

But if I were to pass in a color, it would override the default color, like so:

const pat = new Cat("Pat", "black");

pat.meow();

So once again, if you want any of your function parameters to have a default value that's set when no argument is passed in for that parameter, then remember this default parameters syntax.

# 

# OVERVIEW - ADVANCED ARRAY METHODS

Before we talk about advanced array methods, let’s very quickly review arrays and array methods in general. Go ahead and open your developer console. An array is a data structure that can be stored inside a variable. It is a special kind of Object - it follows some, but not all, of the rules of Objects.

You’ve learned before that arrays come with tons of predefined methods that work on any array.

For example, there is an array method called .sort(). As you might expect, it sorts the array, and its default behavior with an array of numbers is to sort them from smallest to largest. So if you have an array like this:

const arr1 = [3,4,2,1];

And you use sort on it like this:

arr1.sort()

Now if you check the value of arr1, you can see that it has been sorted. [type arr1]

If you were to just type sort() by itself, nothing would happen because sort() is an array method, so it needs to be attached to an array to work.

So that’s what a method is. A method is a function that’s never called by itself, it’s always called using dot notation, on some kind of data, such as an array.

To call both functions and methods, you must include a parameter list enclosed in parens, even if it’s empty, like with sort().

OK, our review is over and we’re now going to talk about advanced array methods. And we’ll start with this: What you need to know is that it is actually possible to pass a function into another function’s parameter list as an argument.

There’s a pair of concepts in programming called a “higher order function” and a “callback function”.

A “higher order function” is a function that takes another function as an argument, or returns a function as a return value. Basically it’s a function that’s written in a way that it uses another function.

We won’t talk about the returning a function concept yet, what we’re interested in here is the taking a function as an argument part.

A “callback function” is the flip side of that. It’s a function that’s passed into a higher order function as an argument, then used inside it.

The array methods you will be learning next, map, filter, and reduce, are higher order functions.

Each of these take a callback function as an argument.

And those callback functions will typically, not always but quite often, be in the form of arrow functions.

Next, what you should know about these array methods map, filter, and reduce, is that none of them are mutator methods. They will not mutate the array they are used on. Instead, they can be classified as iteration methods. Each time you use one of these methods, it will run, or iterate, multiple times - as many times as the length of the array you called it on.

The last thing I want you to know about the Map, Filter, and Reduce methods is that they all have return values. Not all array methods have defined return values, but these do. Map and Filter will both return a NEW array. Reduce will return a single value.

Don’t worry if this isn’t crystal clear yet. Now that you have an overview of some of the concepts, the next three lessons will lead you step by step through how to use these methods, and hopefully it will all become much more clear by the end.

( maybe not the below - too much?) (ended up skipping)

~~It doesn’t matter if the array is in a variable or if it’s an array literal. You could even do this:~~

~~[5,4,3,2].sort()~~

~~And you see that worked. Remember before, we discussed that values in JavaScript can be stored as variables, or they can be stored as literals. Like the number 3 is a literal. If you want to store it in a variable, you would type let x = 3;.~~

~~And you can use them in similar ways. You can say 3 \* 2. You can also say x \* 2. Right>~~

~~It’s the same way with array methods. You can use them on a variable that contains the array, or you can use them on the array literal.~~

# ADVANCED ARRAY METHODS: MAP

* The array method *map*iterates through an array and performs a callback function that you give it upon every single item in the array.
* You can think of it as similar to a for loop. For example, let’s say you wanted to go through an array of numbers and create a new array with those numbers doubled.
* If you were to use a for loop, that might look like this:

const arr1 = [5,21,8,100];  
const arr2 = [];  
for (let i = 0; i < arr1.length; i++) {  
 arr2[i] = arr1[i] \* 2;   
}

**arr2**

* **You could get slightly more sophisticated and use the forEach array method, and that would something like this:**
  + **const arr3 = [];**

**arr1.forEach((n, i) => { arr3[i] = n \* 2; });**

* Or…. using map, you could just do this: **const arr4 = arr1.map(n => n \* 2)**;
* This does the same thing as the for loops, with a lot less mess. [show **arr2, arr3, arr4**]
* Let’s unpack this.
* First, we’re setting up a variable that’s going to hold the new array - const arr4.
* Then we’ve got our assignment operator.
* Then we have the old array that we’re basing the new array on, arr1.
* And we’re using the map method on it. Const arr4 = arr1.map();
* Remember what I said before, map is a higher order function because it takes another function as an argument.
* And for that argument, we’re going to write an arrow function right here, right in the parameter list itself.
* ~~Now, it’s perfectly possible to write the function elsewhere, give it a name, and pass in the function name here:~~

**~~function double(n) {~~**

**~~return n \* 2;~~**

**~~}~~**

**~~const arr5 = arr1.map(n => double(n));~~**

* ~~That works too. But in this case, we’re just making one small operation to each of the array items, so it’s much simpler to just write the function as an arrow function, inline, instead of separating it out like this.~~
* ~~There may be other cases where you do want to separate it out this way, because your function is more complex, or you’re passing in a function that you want to be able to use again.~~
* ~~But quite often you will see this pattern, where you have an arrow function as the callback function right here in the parameter list.~~
* Let’s walk through what map is doing.
* When you use map, a loop begins - a series of iterations.
* And this loop will by default start with the first item in the array at index 0,, and it will loop as many times as the number of items in the array.
* Each time it loops, it will go through the array, grab an item, and throw it into this callback function.
* And this callback function is required to have a parameter of its own. Yes, we’re talking about the parameter list of a function that’s inside the p arameter list of a function.
* **So I chose here to use the letter n for this parameter, to stand for number, since I know I’m iterating through an array of numbers.**
* **Whatever name you choose to give this parameter, it will always have as its argument the value of the current item in the array.**
* That means that at the first iteration of the loop, this **argumen, n, will contain the value of the first item in the array, which is this number, 5.**
* And it will perform whatever’s in the function, then it will place the return value of the callback function, which is 10, as the first item in the new array.
* **Then it will start the SECOND iteration, and this time, n will equal the second item in the array, which is the number 21.** It will double that, and put that value into the new array, as the second item.
* **Then the third iteration, now n is 8, double 8 is 16, so 16 gets put into the third slot in the new array.**
* **And the fourth iteration, now n is 100, double 100 is 200, so 200 goes into the last slot in the new array.**
* Then that’s all the items we have in the original array, so the map is finished, and we have a a brand new array that was created.
* To now test your understanding of map and practice using it, go ahead to the code challenge on map..

# 

# ADVANCED ARRAY METHODS: FILTER

The array method filter takes an array and a callback function that specifies a filtering condition, iterates through the array, and returns a new array that contains only the items that passed through the filter.

Let’s go to the developer console and type out a quick example. Be sure to do this with me.

const arr1 = [1,3, 7, 4, 9,15];

Let's make an array of numbers.

Then let’s say that you wanted to make a new array that only contains the numbers from this array that are less than 7. You could, of course, do this with a for loop, or a forEach. But the most elegant way is to do it is in one line with filter, like this:

const arr2 = arr1.filter(n => n< 7);

You can see that arr1 contains the old array: arr1

And arr2 contains the new array, which has been filtered using this arrow function that was used inside the filter method as a callback.

Let’s trace through it together. We had this array, arr1.

Then we declared a new array, and we set its value to the result from using the filter method on arr1, and we gave the filter method this arrow function.

And the filter method went through each item in the array, one at a time,

and said ok, this item is n, we will check if n < 7 is true, it’s true, so we’ll put it in the new array arr2. Then it got to 7, and 7 is not less than 7, so it didn’t get put into arr2.

Then it checked 4, 4 is less than 7 so it got put inside arr2, but not 9 or 15.

And how that works is, the filter method says, if the result of what is returned from this callback method inside here is TRUE, then the item goes into the new array. If the result if FALSE, it doesn’t.

So that’s a big difference from the MAP method, because with MAP, you will always get back a new array of the SAME length as you put in. Let’s pull up our MAP example again:

* const mappedArr = arr1.map(n => n \* 2);

And mappedArr is the same length as arr1, as you can see

MAP is meant to be used when you want the items in the new array to be different from the old array, but the number of items to be the same.

With filter, you’re also making a new array, but while the length, the number of items in the new array may be different, the values of those items are the same. Do you see the difference?

Of course, there’s always a possibility that your filter could return an array of the same length, if the filtering condition was true for every item. For example, if my filtering condition is: only numbers that are < 100, then all the numbers from the original array will come through to the new array, so the length will be the same in that case. [demo this]

Let’s try one more filter for practice, this time using strings instead of numbers.

Type this into your console:

const animals = ['bear', 'panda', 'penguin', 'osprey'];  
const filteredAnimals = animals.filter(animal => animal.includes('p'));

We'll make an array of animals this time.

Then we'll use the string includes method inside of our callback method to check if each of these strings include the letter p. If the result of this includes condition is true, then we put that animal string into the new filter. And you can see the result here, the new array contains panda, penguin, and osprey, but not bear.

That's the end of this lesson on filter. I hope you can see how useful this array method can be, and be sure to try the Code Challenges for practice.

# ADVANCED ARRAY METHODS: REDUCE

Of the three methods we are covering, map, filter, and reduce, Reduce is probably the most difficult to understand right away, so I advise you to approach reduce with patience to understand it properly.

Let’s start with the differences between reduce and map and filter.

You learned that map and filter both return a new array. Map returns a new array that’s the same length as the old array, but the items inside the array will typically have changed in some way.

Filter will also return a new array, and the new array may be shorter in length than the old array, but the items inside will not have changed their values.

Reduce, now. Reduce will not return an array at all. Instead, Reduce will return a single value.

Let’s look at an example with a numeric array. Please type along with me.

const arr1 = [3,4,13,8,22];

const reducedValue = arr1.reduce((a, c) => a + c);

(while typing) With the reduce method, we need to pass in two values to its callback. We will name them a, c for this demonstration, and that’s to stand for the concepts of accumulator and currentvalue. [annotate: accumulator, current value] Again, the parameters here are not required to be a and c, we're just using those names for this demo.

The currentValue, c, is just like the parameter we passed into the map and filter callback functions, it holds the value of the current array item.

The accumulator variable's value gets updated at each iteration, then eventually, at the very end of all the iterations, it becomes the return value for the entire reduce method.

There is an optional third parameter we could have added here that would set the initial value of the accumulator.

Since we didn't set that initial value, what happens is that at the first iteration, the accumulator will be equal to the first value in the array, and currentValue will be equal to the second.

So let’s trace through it. When the reduce method begins, at the very first iteration, the accumulator is set to 3, and the current value is set to 4. Then the operation inside this callback function here is performed, and we're doing an addition operator, so we add 3 plus 4, which is 7.

And what happens next is that the accumulator variable, a, then is assigned that value of 7. It's as if we're saying: a = a + c, or a = whatever happens inside here, it doesn't have to be addition, it could be any operation.

Then on the next iteration, a is 7. We iterate to the next item for currentValue, last time it was 4, this time it is 13. 7 + 13 is 20, so our accumulator value is now 20. Next, we add 8, then 22, and that’s how we end up with 50. So the way that you can think of A is basically the running total.

it's a different story when it comes to using reduce on an array of objects. With an array of objects, you must set the initial value, or it will not work properly. We will discuss how that works in the upcoming Arrays of Objects lesson.

As I mentioned, the reduce method is typically the toughest of the three we'v'e covered, so don't get frustrated if it doesn't make sense right away. Make sure to try the Code Challenge for Reduce for practice.

# 

# ARRAYS OF OBJECTS

So far, you have been using advanced array methods on arrays of primitive types, such as numbers and strings. We'll now take a lolok at working with arrays of objects. this is a structure that you will often run into.

ARRAYS OF OBJECTS

SLIDE: Creating an Array of Objects Using Object Literals

We'll first look at creating arrays of objects.

Now so far you have been used to seeing objects stored as variables like this:

const dessert1 = {

type: "cake",

flavor: "chocolate",

cost: 4.50

}

const dessert2 = {

type: "pie",

flavor: "pumpkin",

cost: 3.75

}

const dessert3 = {

type: "donut",

flavor: "chocolate",

cost: 1.50

}

This part here, the curly braces and everything inside it, is called the object literal. It’s just the literal value of the object.

When you’re storing a series of objects inside an array, often, you will define the object inside the array using object literals, without assigning each object to its own variable name:

const desserts = [

{

id: 0,

type: "cake",

flavor: "chocolate",

cost: 4.50

},

{

id: 1, // shift enter

type: "pie",

flavor: "pumpkin",

cost: 3.75

},

{

id: 2, // shift enter

type: "donut",

flavor: "chocolate",

cost: 1.50

}

];

It's a common practice to give each object in an array of objects a unique ID. That way, even if the array order get moved around somehow and the objects' index number in the array changes, each object still has a unique identifier.

**SLIDE: Filter Example**

Now let’s say we want to filter for a new array that has only the chocolate flavored dessert items.

But our items are no longer primitives, they’re objects, so you can’t just say something like this:

Const chocDesserts1 = desserts.filter(dessert => flavor === “chocolate”);

This will not work. Instead, what we have to do is this:

const chocDesserts2 = desserts.filter(dessert => dessert.flavor === “chocolate”);

Now if we check what’s inside chocDesserts2, you can see that it only holds the chocolate flavored desserts.

Let’s walk through this. First, we set up a new variable here that will catch the new array that will be returned by the filter. Then, after the assignment operator, we put the name of the original array being filtered, then .filter, then we set up the parameter list for the filter method.

For that parameter list, we pass in this arrow function as the callback function. For this callback function, we are naming this parameter dessert, to stand for a single item in the desserts array, just like before we used n to stand for a single number in a numeric array. Then inside the callback function body, we can use that dessert variable. The first iteration, dessert holds the first item in the array, which is this object with the type of “cake” and the flavor of “chocolate”. Then we check, does this object’s flavor property equal “chocolate”? Yes, so we put it in the new array.

**Slide: Reduce Example**

For the next example, I mentioned in the lesson on Reduce that when you use Reduce with an array of objects, you must include an initial value. We'll look at why that is and how to do it.

We will use reduce to total up the cost of every dessert in this array. Let's type it out first in the console, then I'll step through it. (while typing:) This time, I'll use the parameter names of total for our accumulator value, and dessert for the current item value.

**const totalCost = desserts.reduce((total, dessert) => total + dessert.cost, 0);**

To set the initial value parameter of a reduce function, we put a comma at the end of the callback function, then follow it with the initial value we want to use. In this case, we use 0 as our initial value.

When we do NOT set an initial value, then at the first iteration, the accumulator value is automatically set to the value of the first item, and the current value is the value of the second item. That's fine if we are working with primitive values, but in this case, we are working with objects, and we don't want to set the accumulator value, total, to the first item, which is an entire object, because that is not a number. That would mean we are trying to add this entire object (highlight the first object) to the cost of the second item, 3.75.

In fact, let's actually try that same reduce without the initial value and see what happens:  
const totalCostNoInitValue = desserts.reduce((total, dessert) => total + dessert.cost);

totalCostNoInitValue

As you can see, it gives us this weird result - not what we want at all.

When we DO use an initial value, then reduce acts differently. At the first iteration, it uses the initial value we gave it as the accumulator value, which in this case is "total".

Then it uses the FIRST item in the array instead of the second as the current dessert value. So at the first iteration, we do 0 + 4.50.

The second iteration, total is 4.50, and dessert.cost is 3.75, so we do 4.50 + 3.75, which is 8.25.

Finally, for the third and last iteration, total is 8.25 and dessert.cost is 1.50, so we end up with 9.75, and that number gets returned into this variable, so our total cost is 9.75.

While the initial value parameter for Reduce is required when working with arrays of objects, I want to note that the initial value parameter of reduce can ALSO be used with arrays of primitives, such as numbers and strings. It's not only for arrays of objects - but it's required for objects, for the reasons you've just seen, and optional otherwise.

While we are on the topic of optional parameters, I want to note that both map and filter also have additional optional arguments, and reduce actually has one more optional argument, which we have not discussed.

In fact, many JavaScript methods have optional parameters, and it is outside the scope of this course to go into detail about all the possible optional parameters. If you want to find out more about optional parameters that we haven't discussed, my suggestion is to visit the MDN Web Docs and look up each method you're curious about, as the documentation there will typically have a comprehensive list.

# METHOD CHAINING

[SLIDE: Method chaining means running multiple methods in a single statement by use of returnin values. ... Methods must have return values in order to be chained.]

Method chaining means that you can run multiple methods in a single statement. And the way this is possible is through the use of return values.

We'll look at method chaining with both array and object methods. Let's look at array methods first - map, filter, and reduce. though note, these are not the only array methods that can be chained.

Methods must have return values in order to be chained, and map, filter, and reduce, all do have return values. Remember,map and filter both return new arrays, and reduce returns a single value.

Go ahead and open your console and type with me.

**[SLIDE: Method Chaining - Array Methods with Built-In Return Values**

Let’s use an array of strings this time: We'll use names of states for our string values.

const states = ["Washington", "Maine", "Montana"];

Let’s say that we want to create a new array where we filter out only the states that start with the letter M. We can do that like this:

const mStates = states.filter(state => state[0] === "M")

So now mStates should contain just Montana and Maine.

Now we want to also go through and change every state name in this new array to being upper case. You can do that like this:

const mStatesUpperCased = mStates.map(state => state.toUpperCase());

So what we did here was, we put the return value from this filter method into this variable. Then we used the map method on that variable. Then we put the return value from map into another variable.

We can actually do this in one step instead of two, like this:

At the end of the filter method, I'm going to use a dot, then I'm going to add a map method right here.

const mStatesUpperCased2 = states.filter(state => state[0] === "M").map(state => state.toUpperCase());

And if we check the value of this new variable, we will see that it has the same result as when we did this in two steps, without chaining.

What’s happening here is that we performed the filter method on states, and it returned an array, which means that this whole bit of code here is basically the same as the return value, the array. Instead of stuffing that returned array into a new variable, we immediately just used it as the array for the map method.

Then we put the return value of the map method into this variable.

But we could have actually kept going - there’s no limit to how many times you can keep chaining methods like this, aside from practical considerations of readability and maintainability. You could have added, for example, another map that would add an exclamation mark to each string -

const evenMoreChaining = states.filter(state => state[0] === "M").map(state => state.toUpperCase()).map(state => state + "!");

That’s how method chaining works with methods that have defined return values.

**[SLIDE: Method Chaining: Custom Object Methods and Adding Return Values]**

Let’s look at objects and custom methods that we add to objects.

Here’s a simple bird object with a couple of custom methods:

And it can sing, and it can fly...

const bird = {

sing() {

console.log("The bird sings.");

},  
 fly() {  
 console.log("The bird flaps its wings.");  
 }

}

So we can make it sing to the console:

bird.sing()

And we can make it fly away:

bird.fly();

But can we chain those methods like this?

bird.sing().fly()

No, we can't. And it's not because you can't chain object methods. As the console tells us, it's because we're trying to use fly() on something that's not defined. And the reason it's not defined is that these methods do not have a return value. Rememner methods are really functions and the rule of functions is that they always return something, and if you don't..

And the return value that we want in this case is the bird object, because the fly method is defined on the bird object. How can we do this?

The *this* keyword comes to our rescue. Inside this object, *this* refers to the object. So we can simply return this from each method. We'll update the sing method as I've shown you how to do before:

bird.sing = function() {

console.log("The bird sings.");  
 return this;  
}

And now I can type:

bird.sing().fly();

And we see that the method chaining worked.

We can chain multiple times, even to the same method, as long as the method that you are chaining to has a return value:

bird.sing().sing().sing();

How about the other way?

bird.fly().sing();  
bird.fly().fly();

That still doesn't work because fly() isn't returning the bird object. You can see that without a return value, it can be chained to something, but nothing acn be chained to it. So we have to add the return to the fly method as well,

bird.fly = function() {

console.log("The bird flaps its wings.");

return this;

}

Now we can chain fly to itself or chain sing to it.

bird.fly().sing();

bird.fly().fly()

bird.sing().sing().sing().fly().sing().fly();

So remember this - you can chain methods together, as long as there is a valid return value from the method that you are chaining TO. When you are using arrays methods like map, filter, and reduce, they already have built-in return values. When you are writing custom methods for objects and classes, then you need to include the return value yourself.

# 

# ASSIGNMENT WEEK 1 VIDEO

For your Week 1 Workshop Assignment, you will demonstrate what you've learned about creating and working with classes, objects, properties, and methods.

To begin, open VS Code inside your NucampFolder/3-React folder and create two new tiles.

The first will be an HTML file named **ReactWeek1Assignment.html.**

In this file, you will only need to copy over the code that's provided in the assignment instructions.

This is a simple HTML page that's linked to a JavaScript file.

Next, create this JavaScript file.

After this setup, inside this ReactWeek1Assignment.js file, for task 1, you will need to create two classes, Student and Bootcamp, each with its own constructor.

Then for Task 2, you will need to write a registerStudent() method in the Bootcamp class. I won't go into the details as it's all laid out for you in the written instructions.

Then for Task 3, which I'll demonstrate here, you will open the HTML page you created in your browser. You can do this using the Live Server extension or just open it straight from your File Explorer or Finder, it's up to you.

Then open your developer console on that page. Now you have access to the script file you wrote because it's linked in this HTML page.

Then in your developer console, what you need to do is verify that the code you wrote is working correctly.

To do this, first create an object from the Bootcamp class.

The parameters for this class's constrructor are name, level, and students.

But you will create it by passing in arguments to just the first two parameters, because the final parameter, students, should have a default value that will kick in if you don't pass anything in.

const webDevFundamentals = new Bootcamp("Web Development Fundamentals", "Beginner")

Then type in the name of the object you just created into your console, to verify that it created correctly. You should see an empty array here for the students property.

You could create another one here if you like:

const fullStack = new Bootcamp("Full Stack Web and Mobile Development", "Advanced");

Then I''m going to verify that I can create a Student object using the Student class. You can create it using this example, or you can create your own, it's up to you. The Student class expects a name, an email, and a community, so here we go:

const Neo = new Student("Neo", "[neo@matrix.org](mailto:neo@matrix.org)", "Seattle");

And I'll create two more students

And check their contents to verify that they were instantiated correctly.

Next, I'm just going to clear the console real quick, then I will use the registerStudent() method that was created in Task 2, along with one of the bootcamp and student objects I just created.

WebDevFundamentals.registerStudent(Neo)

And this should give a verification message to the console.

Then I can check the webDevFundamentals object and verify that indeed, Neo is in its students array.

Then I can add the other two students in the same way, and now I can see that they're all in the array.

And if I try to add a student that's already enrolled...

Then I should get a message saying that the student is already enrolled.

So your Task 3 is just to make sure in your console that your code is working and you can create these objects and use the registerStudent() method correctly.

Once you have verified that it's all working, go ahead and submit just the javascript file for this assignment. Good luck!