Objects

Introduction

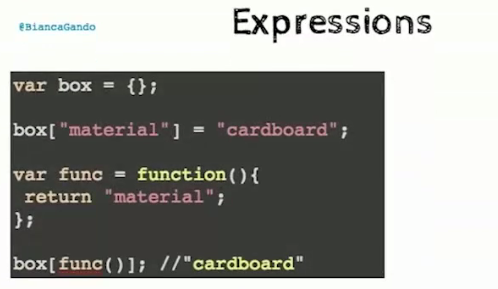
(music) My name's Bianca, I am a JavaScript evangelist for Hack Reactor, which is an immersive JavaScript school in San Francisco. We have some cohorts coming up for our remote program, as well as our in-person program in San Francisco. More details on that at hackreactor.com. And let's see. So I guess we'll just jump in. Most of us have completed the pre-course, the pre-event survey, so I kind of have an idea of your background, what you kind of have your expectations for the class and things like that. Normally I would do a round of introductions. But since we're running a few minutes behind, I'll probably save that for, I'll come around and meet the in person people one by one during exercise time, just for fun. So this is a two part class. Day one, we're gonna do some JavaScript fundamentals. We're gonna go over and deep dive Objects, Arrays, and Functions. And then tomorrow, we are going to go even further and take what we learn today and practice some functional JavaScript patterns with it. So, lots of fun stuff coming up. For those of you haven't seen, we have some links to the slides, that's bit.ly/js102-slides1 as well as bit.ly/js102-exercises, which that's where we'll be going during exercise time. Alright, well let's get started. So first thing is, who is this class for? So maybe you've been studying on your own for a months on Codeacademy, Khan Academy, or more. And you're thinking, okay what's that next step? I've done all the beginner resources, but I want more. And I can't seem to find that middle ground. And I'm not able to solidify and practice those concepts that I was introduced in these beginner tutorials. So that's what this class is for. So as someone whose completed a tutorial before, you have at least two to six months experience practicing JavaScript. And you want to solidify those fundamentals. Maybe you're just looking for more practice. And even better, maybe you're interested in coming to Hack Reactor. And this class was initially designed actually to help people who don't have any traditional computer science experience, get into Hack Reactor. So for those of you who are interested in that, this course is perfect for that. Cool. So like I mentioned before, this is a two part course that I teach in the evenings and on the weekend at Hack Reactor. Part one, again, is just solidifying core JavaScript principles. We're gonna do that using a project for practice. And then part two we're going to be getting started with some functional concepts in JavaScript, as well introducing the Underscore library and how to use it. And then afterwards, you can use this foundation to keep growing as an engineer. Or apply to Hack Reactor, or I don't know. I think the next step is take over the world generally, but you know, whatever you want to do.

Property Access

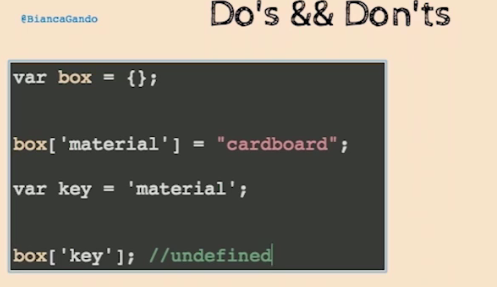
So what are Objects? How many people here have used... Let's practice thumbs. Thumbs on Objects, you've seen an Object before. Everyone's got to do it, even if you don't want to. You have to do it, even Mark has to do it. So what is an Object then? Can you tell me what an Object is? Doesn't have to be a 100% the whole definition. But what's a characteristic of an Object? I'm going to sit here and stare at you until you answer. A way to store data. Yeah, so it's a data structure in which we store data. That's important for a data driven application. We have to put our data somewhere. Properties methods? Yeah, so we can store properties on them. They can be any type, they can be a function. Which would make it a method. Or it could be a string or a number. Or could be another object or an array. What else do we know about objects? Key value pairs? Absolutely, so we have a key value pair. Where the key is a string, usually a series of characters in a word that makes sense to us someway. Blueprint of a class we have online, something that has properties, awesome. Cool, and all those things are right. So an object is a data structure. We have key value pairs. We store our data in there. And the data that we store at those property names can be any type. And just an overview of what we're going to talk about today as far as objects, is we're going to talk about Property Access assignment, Bracket Notation, Dot Notation, when to use the brackets, when to do the dots. What does it mean? We're gonna get into Nested Objects, which is important when you, that's probably how you're going to encounter most of your objects anyways, in a nested data structure. Usually in the form JSON. And you're going to need to know how to traverse them and manipulate them to get that data out that you want. As well as change them and send them back to your API or your backend, or whatever you're doing. And we're going to talk about Iteration and what... Which is just a fancy word for a looping really. Cool, awesome. So what we have here on this first slide, is we have a variable call box, in which we're storing an object literal. So we just have an empty object here. And we're storing in a variable. Cool, whoops. And here we have something called dot notation. And we're just creating a property on our object. And we do this by using a dot property name equals whatever value that is. And again, the value here can be any data type, even another object. And so here's just a visual of what that would look like. So we have our property material here and a value here, cardboard. So my question to you is, what are some other times that you've seen dots in JavaScript? jQuery chains. Yeah, you've used them in jQuery. You guys have typed a bunch of dots in JavaScript before. What about online? Do we have any experience using dots? Like in the URL's? In URL's? Oh yeah dot com, that's true, that's true. But that would be a string normally. Function calls? Who said that? Function calls, yeah so if we have a method, we use a dot. Where else have we seen dots? I feel like these guys in the middle row to the left are burying their eyes, so when have you used some dots in JavaScript? In general. Navigating directory structure. Okay, sure. I haven't touched JavaScript in a few months, so. Okay. Refreshing. Okay, you can open your text editor, look at some old code, command F, just a dot and let me know what comes up. What about in the very back? When have we used dots in the very back? Anyone back there alive? You guys have never used a dot in JavaScript? Have you used a dot? When have you used a dot? In objects and dots to find the value of it. Yeah absolutely, to get the value of an object. Yeah to get value of the object or something like that. Mmhmm, cool, yeah so we can't use weird characters with dot notation. What about in the front? Did you answer the question yet? Some people might use it for an & space. Referencing that that's not much more than traversing an object. Yeah totally. What about here at the stripe shirt? Properties. Yeah, so when asked to assign a property. Ok I said that. Mmhmm, what about like dot length on an array? Who's done that? Dot length. Or what about dot push, dot pop. You object dot keys. So we use dots a lot. And what I'm going to let you know is that whenever you use a dot, you know for sure, unless it gives you an error, whatever to the left of the dot is an object. And so whether that's an array or a function, it's still at it's core, an object. And those syntax rules of dot notation, also bracket notation, which we'll go through soon, stay the same. And so that's important. That's important that all these things are objects. And that we understand that as we're going through this workshop together. Cool. So we use dot notation to add a property. And now we're using dot notation to access a property. Whenever I have these two question marks here, that means that I want you to answer the question. So what about plaid shirt, second row here with the glasses. He's not paying attention to me. I'm sorry. No worries. So what is this going to return? Cardboard. Yeah, absolutely, it's just going to return a string, cardboard. Just like that. And so that would be what we call access with dot notation. Does your slide deck move mine on my screen if I'm watching down there? I don't think so, I don't think so. The URL does reflect what slide I'm on. Can you see that? Yeah. Yeah, so, mmhmm. Cool and so here we are. We are returning that value, cardboard, and we're saving it in a variable, cb. We've all done that before right? Raise your hand, thumbs on that. We've all done that, great, good. If you haven't, I don't know. I don't know what to say. So we just said that this is cardboard, right? So here we are, we're creating an empty object. We're adding a property, cardboard. We're using dot notation to access the value, cardboard. And we're storing it in the variable, cb. Then once we type out cb, and you can imagine that this in the console, it's going to print out cardboard. And what happens if we say box dot material, and we assign it to titanium? We upgraded our box, we're no longer a cardboard box and now we have titanium box. How is that going to affect cb? It's still cardboard. Still cardboard, does everyone agree? Still cardboard, not still cardboard? Okay we have like some mixed feelings. And I'm going to assert that is going to be cardboard. And this is an example of something we call storing things by value. Which we, hopefully, will get to today. But it's just important to note that when we access the variable cardboard here, we're not actually looking up in our code and then referencing box dot material. We're actually looking in Memory, where we had stored whatever value, box dot material's returning. And since box dot material is returning cardboard, cb now only stores cardboard with no reference to box dot material. Therefore, cb here is going to be cardboard, no matter if you change box dot material later on in your code. So to change cb, you explicitly then have to assign it whatever you're going to assign it the new value? Absolutely. And just a sidenote, this is dangerous, but just a sidenote that this isn't true for objects, arrays, and functions who are stored by a reference. So this would be the opposite for that. But I'll hopefully get into more detail about that later. And just to refresh on the dot material, so the second line. Mmhmm. When we're using quotes that's just not when we're using dot notation? So if I wanted to put material in quotes, obviously not with dot notation, but... So dot notation assumes it's a string. So you can assume that material, the property in material, is stored as a string. Which has quotes around it. Got it. Mmhmm, but yeah, this syntax you won't use quotations. Good question. Cool, so we've discussed that box and material's going to return cardboard. What if we do box dot size. What is that going to return? (mumbled answer in background) What is it? Undefined? Undefined, thank you guys for answering that all in unison, it was beautiful. Just kidding. So undefined, why am I writing undefined here with no quotes? Undefined is a type. Yeah, undefined is it's own type. And therefore if we put quotes around it, then it becomes a string. And it's going to mean something completely different. So when we do a property lookup on something that doesn't yet exist on our object, we're simply going to get undefined.

Bracket Notation

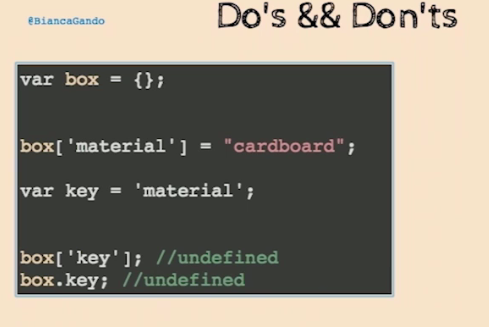
So we talked about dots. Now let's talk about brackets. So dots and brackets are pretty similar, as far as functionality. The syntax is very different though. And the rules are also different. So we're here storing material, property material on our box. And it's going to work exactly the same. So box dot material is more or less equivalent to box bracket quote material. So you can kind of think of these two syntax as interchangeable, to an extent. And so what is this going to return? How about you right here with the green sweatshirt. Cardboard. Yeah, totally cardboard. So we can use them interchangeably, it doesn't effect if we store a value using bracket notation, it doesn't effect our dot notation at all. And then here, what about the green sweater in the second row. What is this going to return? Cardboard as a string. Totally, cardboard as a string. So here we are assigning values. So here's us assigning value. Here's us accessing value. And if we wanted to, then store, whoops... If we then wanted to store a value, we can just put it in cb, and then we can reference it later. Cool. I have a quick question. Is that updating a property or the value on the object? Which one, the second line? Just the brackets, is that updating the property name or the value? So it's adding a property and a value. So it's doing both. Ah, okay. So it's not updating anything, it's just adding it. Mmhmm. Could you use that syntax in the second line there and then reference it with the dot notation later? Absolutely, they're completely interchangeable, totally. Well they're completely interchangeable until they're not. And we're going to talk about when they're not. And so we're going to talk about variables. So when we're going to store our string value, our property name as a variable, we're going to then need to use bracket notation. But the important difference here is we're not going to use the quotations. So we see here in our code that we have our key here, is going to be holding the string, material. And then we then put, in our brackets, that variable name. What's this going to return? What about gray shirt last row? I wish you guys had name tags. Material? Is it going to be material? Anyone want to help him out? Well these are keys right? Yes, these need a key. Oh got it. Yeah, so it's going to be cardboard. Yeah, yeah. So if we just like here said console dot log, for example, key, then that would be material right? That makes sense. You have a question? Question in the chatroom. The difference between box with the curly braces and then if you were to do box equals new object. Hmm, they're different, so one is constructure notation and one is object literal notation. I'm going to recommend that you always use object literal notation. The keyword new is something you see a lot in object oriented languages as well as object oriented JavaScript when you're doing those patterns. That's a little out of the scope of this class. But yeah, always just use this one. And you'll be fine. But they're essentially the same thing. Great question. Okay, so variables. So the important thing again about variables is that we're not going to put quotations around it, because why? Anyone have a hunch on what would happen if we put quotations around key here? Then it's just a string. Absolutely, then it's just a string and it will be interpreted by your browser as a property on box called key. Which then would return undefined, right? Because we don't have a property called key on our object. Cool. And the cool thing about bracket notation as well is that it's actually evaluating it's expressions. So with dot notation, you can't put any expressions or any code to be evaluated. You can't say like, you know, box dot one plus one, that just wouldn't work. However, with a bracket notation we can do that. And here's an example of us calling a function in our bracket notation. So here's a function creatively named func. We call that function. And it returns a string, material. You think this is going to work guys? What's it going to print? Cardboard. I need to start mixing this up, because everyone can just guess cardboard and it's always going to be right, I just realized. So you can go back to the previous slide, with when you did the key thing? Because you used quote. Now if there happened to be a key called key... Mmhmm. Would it then return the value of that quote unquote key or whatever just like... Yep. Okay. Yeah, exactly. Hey Bianca, what are the rules around single quote and double quote? The rules are that you should do as I say and not as I do because I'm kind of not consistent in these slides with it. But be consistent, they have to match one side or the other. Generally wherever you work, hopefully has a style guide. Or if you're contributing to open source, there's a style guide that will tell you which on they prefer. You want it to match in your codebase. But there isn't like a rule. So the interpreter figured that out? That single quote and double quote are the same thing. Yeah, yeah, but for example, if you did key like this, that wouldn't work. But yeah, it's all the same. You got a question in the back? That was the question in the chatroom, so... Oh okay. You just answered it. My slides are strategically created with single and double quotes so that brings up that question. (laughing) It's all planned, mmhmm. Other question is can properties be unicode? If so, how do I quote them? Hmm, I'm not sure, that's a good question. That's a good question, email me bianca@hackreactor.com and I'll get back to you on that. You escape with a slash u, and then the hex code. Oh really? Cool, so the answer to that is you escape it, dash u? Slash u and the hex code. Oh, slash u and the hex code. Very cool, thank you. Could you give an example of unicode? Is that difficult? Um, I don't know, you guys are asking hard questions. What would be an example of unicode? I don't know, we could look it up. Like an ampersand or something? Coefficient of A, which was what? 65 decibels... I don't know. I don't have an example of unicode. But that's something that's an easy Google search. So Herr Manth is the person I should be upset with for asking me silly questions. Okay, cool, so here we go. So, expressions, so our bracket notation, unlike dot notation can handle expressions, and that's pretty...



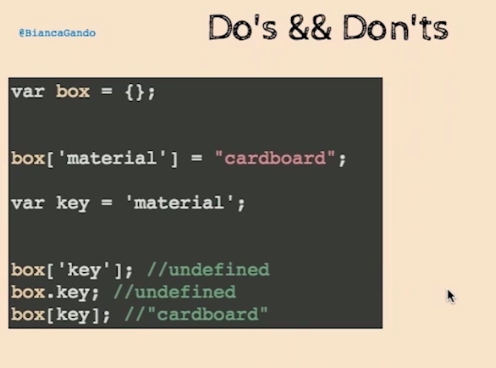
**Object Best Practices**



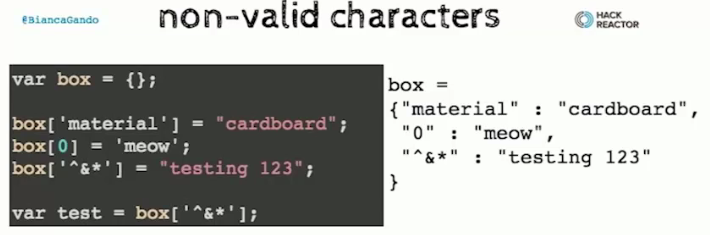
So we have a couple do's and don'ts for our dot notation. What's this going to return? Again the green--these green question marks mean that I want someone... I want a victim to answer. Don't all yell it all out at once. Undefined? Undefined, yeah there you go-- great. What about this one? Material. What is it? Material. Material? No, cardboard...Oh no, that's different. Yeah, undefined. Why is this one undefined? Because it always thinks key is a string rather value key, is key value take it as a key rather than that.



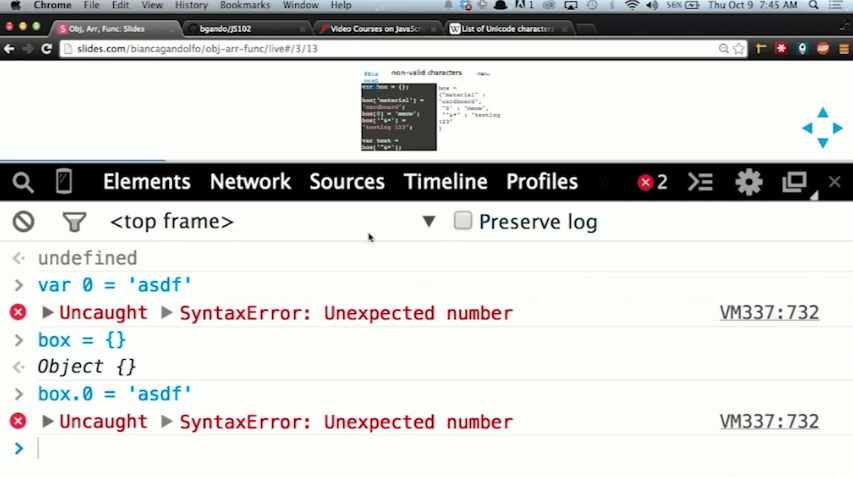
Absolutely, so the reason this is undefined is because the dot notation is equivalent to using the bracket notation with the quotes. So the first two lines are looking up the same thing. They're looking up a property called key, on the object box. Which doesn't exist, so it's undefined. So if you wanted cardboard, you would do the brackets without the quotes, kui without the quotes in the bracket notation? Absolutely. So what's this one?



Cardboard. Cardboard. Very good, cool. So here's some do's and don'ts. Don't use dot notation with a variable. Don't use quotations around your variable. It's going to give you some undesired results. Yeah, if you have a variable just put it in there without the quotes. So what about some nonvalid characters, and numbers and things?



So with dot notation, you have to use valid variable names. When I mean valid variable names, I don't mean a variable name, I mean something that if you type it up in your console here... Can everyone see this? If I said like, var zero equals asdf, that won't work because numbers aren't valid variable names. So that means that for our object, if we said box dot zero, you know equals something, that's also not going to work. So if you can't say var, whatever it is, then that's considered an invalid variable name and you cannot use it with dot notation.



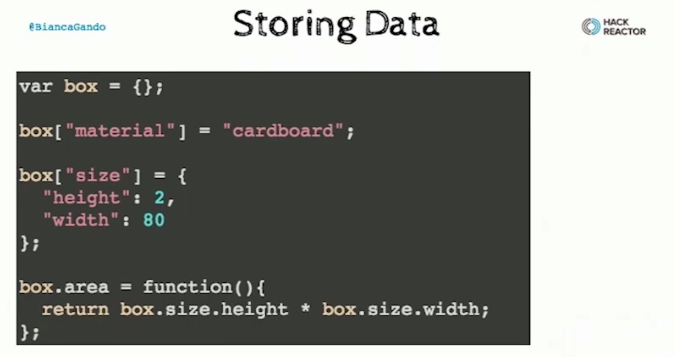
So some things, other than numbers, that are invalid are just weird characters. And the way that we get around this if you have to use them is by using the bracket notation with quotation marks. Unless it's a number, in which case quotation marks are optional. But for example here, we have this really semantic name here, upper carat ampersand star. I name most of my properties like this. It's for readability purposes. And so that's how we deal with non-valid characters, numbers, things like that. And then also if we want to access them, it's the same way. We have to keep those quotations around it with the bracket notation. Cool, and here's a picture of what our object kind of looks like. And I noticed that our numbers, even though we pass it in as a number, it's actually stored as a string. Under the hood, whenever you use bracket notation, it's actually stringifying. Once it evaluates the expression, it's going to stringify those values, even if it's a number. Even if it's an array or a function as well because those are objects under the hood. Are single quotes equivalent to double quotes in all these places? Yes, quotations are super flexible as long as they match. Cool. So here are the rules kind of laid out.



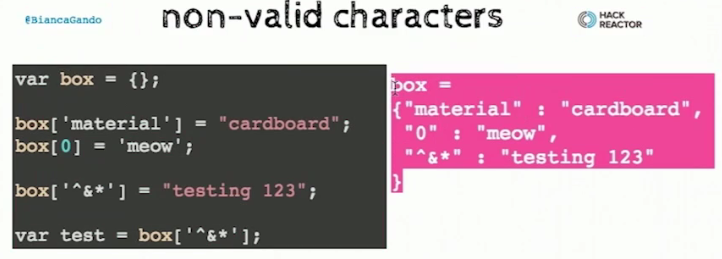
So as we can see, bracket notations, a lot more powerful than dot notation. Dot notation you can really only put strings in there. And remember those strings have to be valid variable names, which means they can't be numbers. They can't have weird characters. They can't be expressions, like math or a function. You can't put quotations around it. For brackets you can do pretty much anything. Some things do need quotes. I have a little key here. So if it has this key here, that means it requires quotes. So if you want a string, it has to have quotes. If you want to do anything with a weird character, you require the quotes as well. And then variables, numbers, and expressions without the quotation marks. Cool, mmhmm? If you pass in a function in the brackets does it have to return a string? It should return a string. Okay. Whatever returns is going to be stringified. The danger with... I don't know why you would really, I don't know, maybe you would put a function. I've never put a function in bracket notation, but if you don't call the function then it's just going to basically like stringify your function. So if you say like var x equals function, it's just going to say like x dot toString. And then it's going to do a property look up on something that looks like function with a quotation mark. So make sure that you actually call your function. If you must do that, but I wouldn't recommend it. It was more of an example that you can do that because it evaluates expressions in those brackets. Does it stringify in the brackets like if it's a number? uh-huh. And so anything can get stringified? Yep. Is there any special string interpolation rules for single versus double quotes? Some languages single quotes is just a string literal. No, it doesn't matter. Yeah it could be anything, really doesn't matter. You guys are really into the quotations today. So many quotes, cool. Alright, so those are the rules. Any questions about these rules? And oh, really really important about these rules is that they don't change. So like I was saying before, everything in JavaScript's an object and because they're objects, these object rules for the syntax stay the same. So keep that in mind. Whenever you see dot notation, wherever you see bracket notation, just remember these rules. And they're always going to be the same.

**Storing Data and Object-literal Notation**

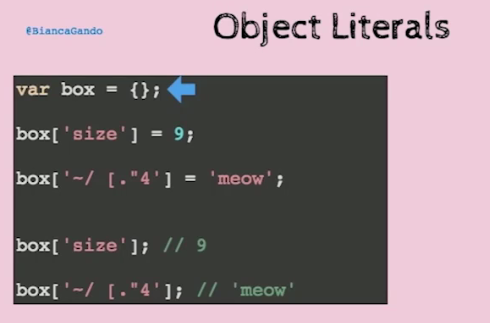
One last thing I want to talk about objects, is storing data. So as I mentioned briefly before, we can store anything in our object.



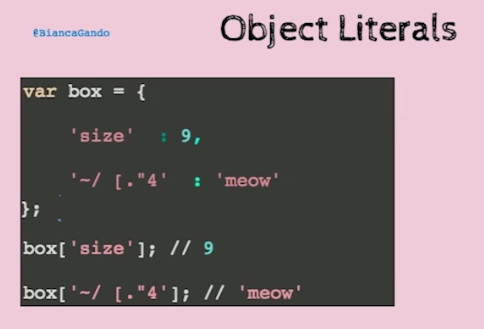
So here we have on I guess the second line here, we're storing a string. But we can also store an object. And then we can also store functions. When we store functions as a property on an object we call it a method. I'm not sure why we have to do that, but that's what we do. And yeah. Any questions about that? Can you say that again? Elaborate. Elaborate, so this slide is just showing that we can store any data in JavaScript. So any data type, so we have strings. You can store numbers, you can store booleans. You can store undefined null. You can also store functions and objects. And when we store our function as a property on an object, we call that a method. So if you hear the term method thrown around and function thrown around, just know that they're the same thing pretty much. Except that a method is just a property on an object that's a function. That's the only difference. But if you really think about it, in the browser, everything's attached to the Window object. So you kind of say that everything's a method. But maybe I'm taking that too far. So area here is the method. Yeah, area is the method. Got it. Mmhmm, cool. And the function part of your lecture. Are you going to get into the whole the difference between a function and a method and all that stuff? I mean yeah I'll talk about it. I mean, the difference between a function and method is there's no difference. Okay. Yeah as far as how you use them, the only thing, it's just a vocab word really that a method is a function that's a property of an object. So if you say like an array has a method, that just means the array has a function, like array dot push. And we put the two parentheses on the side. That means that it's a function. Yeah, that's all, cool. And then just like some styles. A lot of people say, so you can do bracket, you can do dots. When would you use one versus the other? And it really doesn't matter. Again, at your job or if you're contributing to open source, if they have a style guide, of course follow their guidelines. For me, I always use dot notation unless it's a variable or an odd character. Just simply because it's shorter to type and it's just easier. So in general I just use dots, unless I need to do something with an expression. Like evaluate a variable or something like that, cool. Alright, so just going back up to a slide. So we have like two different ways to look at the object, right? We have this sort of notation to the... I don't know if that's left or right. To the side. And then we have this notation where we're dynamically adding properties. So this one right here is called object literal notation.



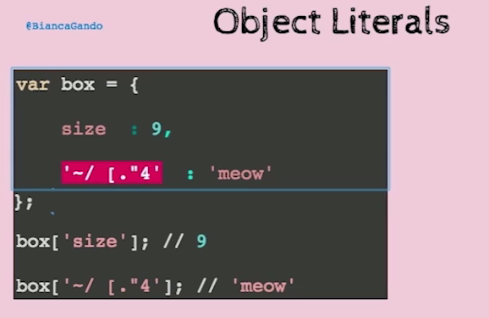
I'm going to talk about how to translate one from the other. So first, again, we're going to start with how we had it.



We have this object literal, it's empty. And then we expand out that bracket. And so now that we have everything inside of our object, we can lose the reference to box, right? Because it's inside, so we take off box. We don't need the brackets either. And then we aren't assigning anything, we're just setting it. So we just have some colons. And then we added a comma, and then we're going to take out that semicolon and keep it at the end. And so that's sort of the walkthrough on how to go from object literal notation, I'm sorry... From dynamically adding properties and values to an object to object literal notation.



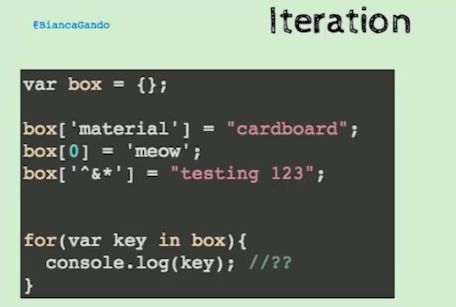
Which you've probably seen both of them by now. And the important thing here is that property names, in this case, are always going to be strings. They're not going to be expressions. But you can put an expression as a value, cool. Any questions about that? Sort of a different syntax. And also in this case, we don't even need to have quotations since it's going to be assumed that size is a string, we don't need quotes. However, if we're going to have something with funny characters, like this,



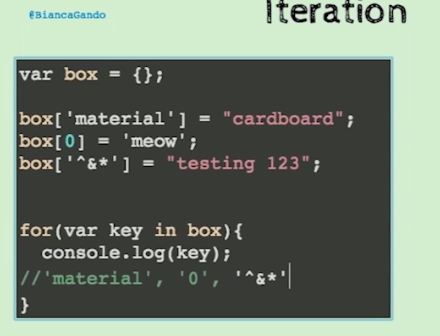
you're going to need to keep those quotes around there because it will confused your interpreter. Cool, and then we can still use our dot bracket notation still the same, even if we're using object literal notation. It's just a different way of writing it. Any questions? I'm going to wait for just like one question. So if someone could just ask me a question, we can move on. So when teams are working, they come to agreement on which way to do things based on a style guide? Or how do you do that? Yeah, I mean, there might be a styleguide. I don't know if it will be as in depth. But as far as like when you should use object literal notation versus adding properties dynamically, it kind of depends on your data and how you're getting it. So if your data's going to change, for example, like after runtime, it might make sense to store in a variable and add it dynamically. Especially if the property names could be different. So if it's going to change after runtime, you would probably want to use the bracket or dot notation. But if it's something that's static and it's just an object that you have and that data doesn't change, then you can just put in object literal notation and call it a day, you know.

**Iteration**

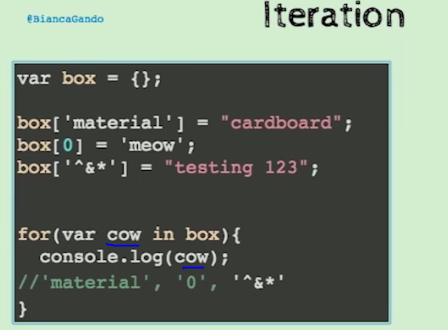
So now we're going to talk about iteration. We talked about adding things and taking things out. What about if we want to just loop through the entire object and get out those values and take a look at that property, so?



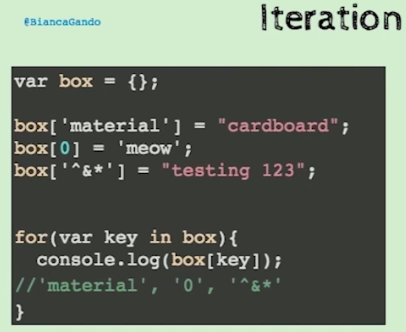
A question came up in the chat room, if size is a string but you assign it a value that's a number, does it become an integer? Wait, one more time. If size is a string, but then you assign it a value that's a number, does it then become an integer? Right, so are we talking about here? Right. This is a number. Right, but if that word be a string there, I think they're asking then if you were to assign it a number... Like this? Right. No then it would just be a string that happens to have a number, a character that's a number inside. I think the question he's asking is, if you set it as a string and then later you set that same property to an integer, it will change it's type. Absolutely. Oh yeah, it will change. I think that's what he's asking. Yeah so if we said box dot size, so we set here as nine. And then we said box dot size = nine, or you know whatever, then we said like type of. Typeof box.size, that would return... You know that would return string. Does that answer the question? Yep. Okay, so back to looping. So we want to loop through our object and look at what's inside. And so we have a handy, what should I call this? I mean, we have a loop, we have a for-in loop. And this loop is special for objects. Which is great, because everything in JavaScript's an object, so it works for almost anything. So here we have a loop that is looping through our box. Some keywords here, for our loop, for this particular loop. There's tons of different loops and well there's like six different loops or so in JavaScript. But this one in particular, there's two keywords to keep in mind, we have for and then we have in, and that's why I call this the for-in loop. And then we also have this var key which is a variable that we just create, and this variable is going to hold the value of the property name always. And that's just the way that this loop works. And then the other important thing to keep in mind is where we have box, and that's going to hold reference to our object in which we're looping through. Right, so we have our box up here. And we're looping through here in this box. And so, knowing what we know, about this loop, what do we think this console log is going to return for us? Material, zero, funny characters?



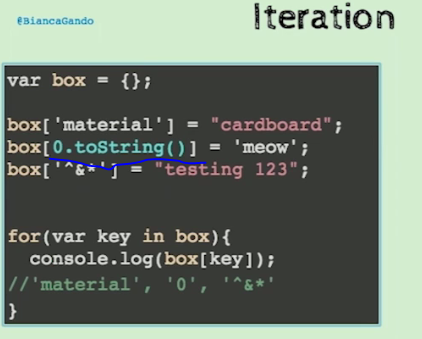
Zero, you mean you can't pronounce this? I think it's pronounced krrrrrm. I think that's how you say it. Rrrrrr. Rrrrrm, yeah. Cool, so it's going to log our property names, cool? So what if we wanted to... But not necessarily in that order. Yeah, not necessarily in that order. There's no guaranteed order in an object, so it will come out however. What's a little confusing there is that... Is key like kind of a reserved word or could you have called it anything? No, we could call it anything, it's just a variable.



Right, okay so it's not like you know as long as it's consistent. Yeah and so we tend to say key or prop, but that's just a convention, it doesn't actually mean anything. I usually just call it cow in my code. I'm just kidding, I don't. Would it still be valid if you did the variable declaration outside of the for expression there? And just passed the variable. Mmhmm, cool. Now why is that on the key and not on the value? How does it know that cow is referring to the key? Because that's just how the loop is, so that variable, whatever variable you put here for the in, is going to be the key. And so you use dot or bracket or whatever for the value if you wanted to loop out cardboard testing? Totally and that will be my next question. Let's see, who here is really shy? I'm going to pick on you. What about, you look particularly shy this morning. If we wanted to print out the value. So just like Rich in the front row said, if we wanted to print out the value how would we do that? So if you wanted to do cardboard, meow, testing 123, how would we do that? You're asking me right? Yeah asking you. Yeah I don't know, I was literally just playing around with that. Trying to get that exact same thing here in my editor, I don't know. Cool, that's fair. What about you over there? Box bracket cow bracket? Cool, so whoops... So box bracket cow. Will that work? Yeah. What about this? What about purple shirt front row, is that going to work? Yes. Why? Because bracket is the same as dot. It's close except for when? When it's not. Yeah, when it's not. (laughing) Except for when it's a variable or a special character or number. So in this case, actually cow is a variable. We can see that here actually when we say var cow. We see that that's a variable. So in this case actually, we can only use bracket notation for it to work.



Thumbs on that? Everyone's got to have their thumbs. Thumbs in the chat room, so a lowercase b is thumb up and a p is thumb down. I know, chat room doesn't like thumbs up. Question in the chat room, in this for loop, he was just asking if you should be checking box dot has own property, before locking it? Yeah and that's why when we get to our functional program, we're going to talk about looping using underscore and why it's better, because you don't have to do a check like that. But yeah that's a good point. Cool. Bianca? Mmhmm. In that second assignment box bracket zero bracket, I'm still struggling with that. Uh-huh. What is that actually doing again? Uh, this one here? Yeah. Yeah, so what that's doing is it is creating a property, zero. And once you pass it in there, zero's a number. So you don't need to put quotes around it, but underneath it's going to be running two-strings.



So you can kind of imagine it. To say like this. So it's not looking at a zero at the index and returning that material property? That's what I'm struggling with. Oh, sort of like, you're thinking of like on an array. Yeah. No, no it's different. And then also with an array, we'll talk about that after the first exercise. That these things still work for arrays and that the zero with property, in an array, is actually just a property name. It actually isn't really paying attention to the fact that it's the first one or not. It's actually like zero colon, you know, value. In this case, meow. But yeah, it has nothing to do with the order. Mmhmm? They're just asking if you could explain again why dot notation wouldn't work for that console dot log? Sure, yeah so dot notation wouldn't work because key is a variable name. And since it's a variable we have to always use bracket notation, because we want our interpreter to look up that variable instead of just using it as a string. So if I did put quotes around it, what would happen is we're looking for a property now called key on box. Not in the variable and so the variable, key in this case, in this loop, is going to hold different property names. For all the property names in the object each time. So this object has three properties, so it's going to loop three times. And every time key is going to hold the value of the property name. So the first time it might hold material, the second time it might hold zero, but it's going to be stringified zero actually. And then the third time it's going to hold srrrm. That's how you pronounce this third one, I think. My accent's kind of off, but, yeah. Do you use var on the for loop since it already know that's the key? You should always use var, because it keeps it from being a global variable. But it wouldn't break it if you didn't do it, but it might interfere with other. If you have a bunch of for-in loops and they all have key, and they're all global, it could do some weird things, potentially. Cool, sure. I actually get three undefined's when I run that. Oh really? Oh with the quotes around it? Yes. Yeah because we don't have a property key at... In the box. In the box, so the quotes around it is an example of how you shouldn't do it. So without the quotes is an example of how you should do it. Great. But yeah, why do you think that there's three undefined's though? Because it's going through each property. Totally. Each one. Even though we're not doing anything with those properties, it's still going to loop three times. Even though you're looking up that quote. Because in the box it doesn't know if there is one called key, so it's got to go through it to look for it. It's just going to loop, no matter what it's going to loop three times. Because that's just the mechanics of the loop. It's going to loop through all of the properties on the object. You're iterating all the properties, but you're always looking for the same thing. Right but if we're looking for key in quotes... Mmhmm. How does that loop know that there isn't? The loop doesn't know anything. Right, so it goes and it goes to the first property and it checks to see if that one is quote key, because it's undefined. So it has to iterate until it finds something. Right, but you're not comparing the key, you're just iterating all the properties to do something with whatever that property is. Yeah whatever you're doing in this block... Oh got it. The loop doesn't know about it, it doesn't care about it. If you called it quote key or quote hello world, it wouldn't matter. Right. Any more questions? I think that another question, are you also using bracket notation because one of the properties is a number, or is it just because of the variable? Just because of the variable. Just because of the variable. Because again, just one last thing, if we put quotes around it, or we use dot key, it's only going to be looking for a property called key on the object, which we don't have, right? And that's why we need to have the brackets.

**Objects Exercise**

So now it's exercise time. Normally at Hack Reactor, how we handle exercises is we would go to the pairing machines and do that. But since we can't do that, we're going to have mix it up. If you're here together, I recommend pairing, I think it's really helpful. And it makes things fun. You learn how to communicate your code. And actually you can clarify in your mind your understanding to take these intuitive understandings of how JavaScript works. And apply these rules to them. And really make that solid. So if you can, pair up. And I'm just going to walk you through a little bit, how to get started with the exercises. Again, if you just type in the bit.ly, so the bit.ly is bit.ly/js102-exercises. And you can also find that on the first slide. And I think Mark sent out an email prior. And so that will bring you to this GitHub repo. And what you can do then, is you can download, hit this button here, download zip, whoops. So you download the zip here. Let's see. I'll just save it for now. And so what we're going to be doing today is we're going to be doing this eFarmony project. So eFarmony is this hot new dating app for animals. You know, looking for fun things. So if there's any FiFi's out there, it's a great idea. And I have hundreds of engineers across the world working on it right now. Jokes aside, we're going to come down to repo and we're going to start here. In the Before You Get Started section, it's going to tell you sort of how to navigate the files and where to go. So you're going to... We just downloaded the zip. And now we're going to explore the files. So if we show this in our finder, oh we have to un-zip it, of course. And then we're going to go in here. And so the first thing we can do, is we're going to open our index to html file. And we'll notice that this page is intentionally left blank. And basically what this is for, this html page is for, is just to be able to play in your console with the code that you write and your script.js file. So if we go here to our script.js file, and we're going to open it with our text editor, whatever you choose. And we'll see in the comments here, we're going to use this file. It's actually not just part one, it's going to be... We're just going to do all of our code in this file. And so if we just console.log, animal here. You're going to be typing animal a lot, and farm a lot today. And we refresh here, we see we have a console.log animal, so that's one way, that's one workflow again. You type in the text editor, you save it. Saving is important. And then you refresh the page. And that's when you can access the changes in your code. And then, for example, we can say var myAnimal equals an empty object. And we can give it some properties. You know, maybe it's a sheep, so it says baaah. Again I saved it, so for me, command S is save. And then I refresh the page, command R. And then I can type in myAnimal. So notice myAnimal, I can then access in my developer tools. So I'm using Chrome, you can also use FireFox. If that's what you prefer. And to open your developer tools, you go to View, Developer, and then JavaScript Console. And if you get lost in it, you can always on this first... There's a bunch of tabs and it's easy to get lost in the tabs, but you can always just click here and click back to Console. And so that's the workflow of how you're going to be doing this project. If you're online you could try JS Bin. It might be more helpful if you want to share your code in the chat room. So if you put it in JS bin, you would just need to put your JavaScript here, and then you'll have your console. So if you open your JS Bin, this way. So where you have the JavaScript tab on one side and the console on the other, and then you can do the same thing. And then you would hit Run. And that would print out to your console. And so that's sort of the workflow of how you're going to be doing this project. So throughout the day what we're going to be doing, is we're going to be implementing two parts of this app. We're going to be making the data structures and we're going to be adding some functionality to it. There's no UI, this is not like a jQuery html CSS class. It's only going to be two parts of your application. Again, the data structures and the functionality. Like very crude, basic functionality. And if you think about it, apps are really just three parts. You have your data, you have your view, like the UI, and then you have the functionality behind it. So you're kind of making two-third's of an app anyway. So even though it might not be like flashy, that's two-third's done. And if anyone makes a UI for it, please send it to me, that would be really cool to see that. Cool, so we explore the files. Oh some other fun things, is I do give free tutoring to people who correct... Whoever sends the most corrections to either my slides, my exercises, or clarifies any unclear directions or gives me anything like that, I do give free tutoring for more directions. And you can see this tab,a note on free tutoring. And yeah, and so what we're going to do, is we're just going to do the objects one. You can open that here. And this is just a ReadMe file, so you don't have to download it or anything, you can just go through the project. And I'll give you about 15 or 20 minutes or so to go through this, and I'll just walk around.

Objects Exercise Solution

So, do we have any questions about... Yeah Mark. Yeah, this one is on the chat room. At the beginning, so I'm not sure if they answered it themselves, but what means tagline in an animal object? Like any string. Yeah, it could be any string, it could be any string. Like your spot for animal dating, that's what you were looking for. What's that? Like your go to spot for animal dating, that was the tagline. Well it would be like your user tagline. So on your profile, I don't know, like under your name it could say like... I'm the hottest sheep in the farm. Yeah, something to that effect. So you know like, single sheep looking for love. You know, something to that effect. Into other animals. What? Into other animals. Yeah, into other sheep, sometimes cows, you know. Just kidding. Okay, so let's just go through the solutions. I want to thank Katrina, who is my awesome Lead TA at Hack Reactor for providing the solutions. She is awesome. So the first thing is we're just going to create an empty animal object. So we're just using that object literal notation, storing in a variable. We're giving it a name, so animal.username. We're going to call it Mittens, and it's going to have a tagline, which is pet me. And so this is dot notation, this is bracket notation. Again, at this point, they're interchangeable. So here we're creating an array. I know we haven't gotten to array's, but I'm assuming that you've used them before. An empty array, and then we're assigning that empty array to animal.noises. So when we console log our, one second here... When we console log, our animal object, we see in our console that we have an object. Should print out something like this. We have an empty array. With a tagline and a username. Thumbs, everyone get that far? Cool, is everyone being honest? Cool, awesome. Thank you everyone. So for the counting one, here we are, we're going to loop through all of the properties in the object. So we have key, and every time we loop, we're going to increment our counter. So cow plus plus is the same as saying count equals count plus one, it's just a shorthand. And here we're going to do that check. I'm going to recommend that we always use triple equals. Can you explain that a little bit? That's one thing that I've never really understood. Triple's versus two-quils. So if triple equals is a strict check. So and then double equals is more lenient. So like for example, if we do string one double equals one, that might return true, even though that might not be what you're looking for. And then because JavaScript has some weird like falsie-truthie bullion things it's better just to always do a strict check, with the three equals. Could you argue that in some cases, you want to use double equals, because you might not be sure what the value is or something like that. Yeah, I mean hopefully... Like should you maybe if you're developing an app, should you try to force yourself to using this triple equals all the time? And might be sure that you know... Yeah even though JavaScript's really flexible, you should try to be strict. Okay. And like kind of know what you're expecting. You don't want your app to give you something that you're not expecting. Yeah that makes sense. Because who knows the consequences of that. So try not to. If you really need to for some reason, then if you're working with some other code that you don't really have control over, and you know what you're doing. Thanks. Mmhmm. Some of the type convergents, can you variable how the browsing manufacturing as well, so you might add the same... That's also the point. On pages or across browsers. Absolutely. I just wanted to clarify something that variable return from the for-in loop, it's always the name of the property not the property itself. Variable return from the for-in loop. Well there's no variable here being returned. Just so like, for example, if I return to variable here. So what key gets set to there in the for-in? Oh what key is at any given moment? Yeah, it's the name of the property, not the property itself. Yeah, so it's the property. So if we have like a property name, and then a property value, and so the key is always going to be a property name. If you think of an object as like a filing cabinet or a dictionary, or some language Python, call them dictionaries. You call it dictionary or hash. You can think of it as a filing cabinet where we have the files and then there's a label. So the label in which we're going to look up our files, is going to be property name. And then what's contained within those folders, is going to be the value. So for example, if you want to look up the username in your filing cabinet, you would have to say fillingcabinet.username. So that means you're going to be opening the filing cabinet, looking through, selecting that file whose tab says, I don't know if people even know what filing cabinets look like anymore. But whose tab says username and you pull up that file and you open it, what's contained in it is the value. And that's like the metaphor I like to think of, for an object. So, and the key would be that the property name, which is that tab, cool? Awesome. I just wanted to make a comment on equals equals. Mmhmm. That's basically allowing quote conversion and cow, since you don't know JavaScript, it covers a lot about type origin in JavaScript in it's fourth book in the series called Types and Grammar. Oh cool. So it's like I'll get to play stack up, if you really want to learn that... Awesome, cool. So back to the exercise, so we have this for loop and here we're going to count, excuse me. We're going to count how many times we're looping through. And we know that whatever this number is, it's going to be the number of properties we have in our object, right? So every time we loop, we're going to increment the counter. We're also going to do this if L statement. So it's always going to check if the key, that property name, equals the string username. If so, it's going to come into this block and console log, hi my name is, and then name. And so notice here, we're accessing the value at that key. Versus on this line, we're accessing the property name. So important thing and we're doing bracket notation again because key is a variable. And just remember in loops, when you have like var I equals zero in some loops or key, those are always going to be variables. And so that should be a cue if you are looping, that you're going to be using the bracket notation and not the dot notation. So in here, the same thing, except if it's a tagline, you console log, I like to say, and then animal key in this case it's pet me. So a follow up question is, what happens if you return, hi my name is, plus whatever, instead of using console log inside the loop? What's going to happen? Anyone try that out? Did you guys get there? Who got to that point? Okay, can I see thumbs on who got to this part? Okay, so most people didn't, okay. Okay, well the answer is that when you put a return statement inside a loop... Actually I don't know if you can even put a return statement outside of a function. No. It was broken. Okay, cool. But if this was inside of a function, and you put a return statement inside the loop, what it would do is it would immediately stop looping and return out of the function. We'll get into more detail about that when we get to functions in a few minutes. But that's something important, that's why I have you console dot logging and not returning, because it would stop the loop.

Arrays

Arrays vs. Objects

(music) So. Array's. What do we know about Array's? Container. Array is a container. Just like an object, it's a data structure, right? (voices speaking at once) Yeah so we can contain, we can have an ordered list in our array. You can nest them. Yeah, you can nest them. You can save any data type as a value in your array just like an object. Anything else? An array is an object? Yeah, an array is an object. If you're coming from a object-oriented language, you could think of an array as a subclass of an object in that it inherits from an object. If that doesn't mean anything to you, don't worry about it, but that might sort of clarify for those who sort of have that background. So we have arrays. Arrays are a data structure, they're another data structure that comes out of the box with JavaScript. We tend to use them for things that are ordered because of the numerical indices give us just a natural way, you know, because numbers are ordered, right, it's kind of a natural way to keep things in order. Versus objects, when we tend have to string properties that don't have an inherent order. But at its core, what's important to note is that an array is an object, and so the rules are going to be the same. And we're going to explore that, and then we're going to talk a little bit about, we're going to talk a lot about how arrays are objects, and then we're going to talk a little bit about how arrays are different than objects. Cool. So just an overview of arrays v. objects, we're going to be talking about access and assignment, we're going to be talking about native methods and properties. When I say native, all I mean is it comes with the language. Because there are some methods that you get from libraries, such as the underscore library, which we'll cover tomorrow, but there's also methods that just come out of the box. You can just open your console and say, like, array, whoops, array dot, and then, actually that's not working. And then you can say array dot and we have all of these methods that just come out of the box. And some of them you might be familiar with. Right. And then we're also going talk about iteration techniques and how they're different for arrays and the same. So when would we use an array versus an object? Like when would you choose to use one versus the other? Yeah so if you want it to be ordered. Doesn't have named properties? Yeah, so if it doesn't make sense for it to have name properties, like maybe, maybe you have, like, like a bunch of names, you can say var names equals, and then you could put like, the names of. Here, let me. Put that up. And then you can put like the names of your pets. You know. Whatever your pets are. Does anyone have a pet? Pet name? No one. Wonder woman? That's a good one. So, so this just makes sense, right? For it to be a series, can you guys see that in the back? A series of, because it's all the same thing. They're all names, so you wouldn't want to say, like, var, you know, names, and then have an array and just be like name1, right. That wouldn't really, that's not adding to the data, that's just adding extra data that's not important. So maybe they're all the same thing. You don't need to key value pair, it's ordered, what else? Maybe you want to use the array methods that come with it? Like maybe you want to be able to sort, maybe you want to do something with the length property, what else? That's all I can think of right now. But yeah, so there are certain use cases for arrays and objects. And so you need to choose wisely.

Access and Assignment

So here's some examples on how you've probably used arrays in the past, so here we have our box. Notice the difference here is that we have the brackets and not the curly braces, and we have box index 0, whoops, is true. So we're adding a Boolean there, that's why we don't have quotes around it, because true is a value in and of itself, it's not a string. And then add the second index, or the first index, I'm sorry, we have meow, and we can always use box.push to add something at the end in here. we are just adding an array. I'm sorry, an object here. And then if we wanted to use i, we'd probably use i before when we're using a loop, the for loop with semicolons. And you could just use i here. What do you think this is going to equal? Can everyone see that? It's kinda small. True? True. Right, because i is going to do a lookup, and it's going to be 0, and so we can imagine that I substituted there, 0, and then that's true. Cool. And then, a 1, we know is what? meow. And then, what if we box.pop? What is that going to return? True? Gonna return meow. Actually, it returns, think it returns the thing that you remove. So what is it going to return? Hello goodbye? The object. Yes, it's going to return this object. And then, and then if I said console.log, console.log box, what is box going to look like now? True again? Yeah, so what pop does is it take the last thing off, and returns it, and so after that, we just have two, two things in the, in our array. Any questions about that? So we've all seen that before, right? We're using bracket notation, we're using push and pop, we're using variable i, all things that we've seen before. Cool. So, cool, we're comfortable with that. Now let's talk about how our, our array is similar to our object. And so this is just our box from the previous example, so we have box as an object, and then we change one thing. See that, did you guys see that change, it's a little subtle, it's the first line, really. We're changing those curly brackets to square brackets. Now let's explore what this does to our data structure. So, when will we, what do we predict, what do we predict this is going to do here on second to last line? Nine? Nine. Anyone else? It's undefined. Oh. Bianca? There's some questions in the chatroom, and I have one too. Can you go back one slide? Sure. So underneath box.push, if you did another, I don't know what the word for it is, but another assignment to the box object, like box And then you did that same box.pop at the bottom, would you still get what you pushed? Or would you get the last thing? You always get the last thing. So this push would be, this would be at 2, so index 2, and then we add something to 3, and then if we popped, it would then be the last thing. So pop always does the last index. OK. Mm-hmm. And it returns the last index, and it's destructive, so it takes off the last thing, and that's important to note. So any other questions? OK. Oh, as to that one, I think you just answered their other question, but, can you then reference those different copies? Oh they're actually, so they're at the same, JS doesn't actually alter the original array, it just makes copies, and modifies those? I'm not sure. So I can kind of show, var x equals this, so, so here's an array, so we x.pop. So it's destructive, so it actually is deleting. But if you were to assign like that array x to another, like y. Ohh. It's passed by. Right, I see what you mean. In that case, yeah, that is a copy. Yeah I see what you're saying. So if you did, y = x, right so we have y and then we say x.pop, and then we look at x. And then we look at y, oh no it is the same. It is destructive. (several voices at once) Yeah, yeah, so arrays and objects are passed by reference, but, yeah. OK. Cool. So, here we go. So some people said it was undefined here, box oops. Any other guesses? Who thinks, who thinks it's going to be nine? Who thinks it's going to be undefined? OK, a few people. Well the cool thing, and maybe the confusing thing about this is that it's actually going to be nine. Because an array is an object at its core, that means that you can add properties to it. And you can kind of think of, the array like this. So we have an index 0, called meow. And then we are, or property 0 you could even call it. And then we have size 9, so you could think of, of your array as really just an object, that also has numerical indices. So what's box at 0? What about Kim? Nine? Box at 0? Nine? It's a good guess. So that's a common confusion, that since we put size in first, that that would be at the 0th place, but it actually has nothing to do with it, since our bracket notation is the same as when we look up on an object, it's looking at that property 0, and it's going to return us meow. Whereas size here, if we wanted to look up size, we just have to reference it by size. So how does it order it, then, if you're allowed to put-- We'll get to that soon. Oh, OK. Yeah, no problem. Cool. Is that clear? So even though size is added on first, it's not at the 0th index. It's actually just a property on the object called size. Stepping back, how prevalent is this in actual coding? Somebody using an array versus an object? You don't, you use it indirectly. These rules you use indirectly, like for example, we're going to get into it a little more, but we have different properties and methods on arrays that we use all the time, and then it goes further when you start diving deeper into JavaScript, and you start talking about prototypes and things like that, it becomes really, clear why it's important to understand these underlying concepts of the mechanics of, of the array as an object. Cool. All right, what about box.size? Nine, yeah. So the rules are the same, so we can use dot notation, we can use bracket notation, just the same as with an object. Yay, the rules don't change! Aren't we so excited? That makes things a lot simpler, right? Or not. Simpler? Simpler? Not simpler? Middle, OK. Well, maybe in time, you'll feel more at ease with this concept. What about, so we already said box.size is 9. What about box.0? What about bin? You can't do that, right? Right. Cause it's a number, yay! Good job. I usually trick people with that one. What'd you say, not in Minnesota? (laughs) Is that where we are? Today. Today...cool.

Iteration

So those are some things that are kinda cool about arrays, that make them just like objects. And we were just talking about this loop, this for in loop, that we can use with objects. And if an array is an object, shouldn't it work? What are our predictions? Is that a prediction? OK, I'll get to you in a second. What's your prediction, what about, let's see. What about Kevin? Do you have a prediction? Log the values? The values? Yeah. Or the property names? The values. The values. OK. So we have a guess for the values. What about Andy? The property names. Property names. If the rules don't change, it's property names. Yup. So it's going to be the property names, because the rules don't change. And if you don't believe me, you can try it out. And the rules don't change. Check this out, our 0th numerical index is actually a string. That's interesting, right? So the rules don't change. And we can, we can, oh did you have a question, Mark? I'm sorry, I forgot. Yeah, do you have the chat up? It might help just to look at questions and double back now, but any difference between, and they have a few examples there. So the question is, is there any difference between... Here, I'll type it out here on the screen so everyone can see. box2 and then, so we have box.asdf = true. and then box2.asdf = false, I'm sorry, equals true. And this says, is there any difference here? And I'm going to argue, no that there's no difference between how we're adding a property and a value to an array versus an object. I'm going to argue that that is exactly the same. Even though it's an array. Cool. So we have this loop, and this k is printing out again, just the property names. So how would, if we wanted to get those values out, how would we do that again? What about, Grace? You would add a reference group with brackets. Absolutely. Cool. And so that's going to then log our values, which would be 9, and meow, right? Thumbs so far? Thumbs, thumbs in the back? Thumbs. Cool. All right, what about this, what is this going to log? Anyone? Nine and-- The values. The values? Good guess, but remember that whole variable dot notation rule? Remember that rule? Thumbs if you remember the rule? That you can't use dot notation with a variable? Thumbs? Thumbs? So since k is a variable, we can only use bracket notation. I'm just going back to this rule that we discussed earlier. So this is going to be actually undefined. Undefined. Because k is a variable. And if we go back to that slide, the rules? all the way down down down down down? That for, dots, we can't use variables. For brackets we can. Here, so. Cool. And we know, again, we know k is a variable because we just see var here, right? For var? That means k is a variable. Cool. So this is how we would do it then. Here we have the bracket notation. Cool. Cool, so. That's great. It behaves exactly the same as an object. Then how come we've never really noticed that before? Like when we loop right, we don't really use the for in loop because we're not really interested, and usually we're, if we're using an array, we're usually not interested in those named properties, so even if we did have this size here, this size property, we're probably not too interested in it, and if we were, we'd probably choose to use just a vanilla object. So how are we going to loop? We're probably going to use a loop like this, right? Has everyone seen, thumbs on this loop? On this for loop with semicolons here? Cool. So this for loop with semicolons is pretty interesting. Remember when I said the for in loop was like a thing that was made specifically for objects? That's not true for this for in loop. This for in loop is just mathematical, if you look at it all, how you give directions to the for loop is just this signature right here. We have for and we have var i = 0, so that's saying, create a variable, i, start at 0, and then it says loop, while i is less than the length of our box, right? And then this one says increment i plus 1 every time. So every time you loop we're going to increment i. And we could change that, you know. We can do plus equals 2 if we wanted. If you want to do every other one, or every third one. Right, you could change it. But i++ seems to be the norm. You could also count down, right? i-- But the mechanics here is that it's, they're just numbers, right? So if we console.log i, what is that logging? 0, 1, 2, 3? 0, 1, 2? Yeah. 0, 1. 0, 1. We're going to get to the length property in a bit, but the point here is that i just a number. Like, i doesn't know anything about our array. The only thing it knows about is this number here, box.length, which we could really just change to 2, and it would be equivalent. So there's no relationship between this loop and the array. It's just like mathematical directions. And so if we say increment until 2, it'll just console.log 0 and 1. Thumbs on that? Cool. What about this? Actually, hmm. I have a question. Sure. So, why is length 2 and not 3? We're going to get to that in just a second. Yeah. Let's just do this. So, if we wanted to access a value, could we do it this way? No. It's a variable. Yeah. I'm going to drill this into you until everyone, like, hates me for it. So here we have again, just our bracket notation with a variable, and maybe we're starting to see that i is just a variable, and we have to use the bracket notation only because it's a variable, not because the array is an array, and there's some special syntax surrounding it, it's simply because it's a variable, and also if we go back here, to box at 0, for numbers, remember, numbers we always have to use a bracket notation. And that's why you tend to see bracket notation with arrays, because we tend to use them for their numerical indices, and bracket notation just makes sense. So when I loop through that, my output is meow, and then whoohooo, so what index is the size or is the value 9 at? Yeah, so if you look down here, you can see, kind of think of it this way. So we have 0, we have 1, and then we have size. It's not at an index. It's just a property on the object. Oh, it's a property. Yeah. And so, since we're looping until 2, and i is a number, it's just going to be and i is remember going to be 0, 1, so it's going to say box at 0, right? Which is meow. You can think of it like this too. Or, box at 1, which is going to be whoohooo, because we pushed that onto the end, right? And so we see we have this property 1, with the value whoohooo. Does that make sense? We have size with the value-- And then we have size, which we don't touch, right? Because i is just a number. All right, can I have one question from the audience? Why would we use arrays instead of objects when it's pretty much the same thing? So why would we use arrays instead of objects? Because arrays have methods on them that deal with order, and you can sort them, you can slice them in ways that you can't do for an object. Objects are better for other things, like quick lookups, or having data with like, that is categorized, like if you had like, a user, just like your animal user, it has different categories on it. You know, you have a username, then there's a value there, and you have like maybe an age, and then it has a value, versus an array, if you tried to represent a user data as an array, it likes the name and then the age, it just, it's hard to keep track, you'd have to keep track of like the index, kind of. Like if you're using an array in a traditional way. Aren't, typically, I guess it depends on the processor, but iterating over an array is faster? Is that true? You know, I'm not sure, I haven't tested that. Well there's like a test thing I found, and in my browser, it is. By like, quite a bit, actually, so. Interesting. I don't know if that's true. I guess it depends on the browser, and how they implemented their engine or whatever. Yeah, that kind of makes sense to me. Cool. So what is the like then? Mmm, let's see. How many, I'm almost there. OK, wait, I actually am there. You just, we're coming to this. So the cool thing, so I'll just jump into this unless we have any more questions about iteration. Think this might be the answer. So there are a couple things behind just you know, but, so, if i is less than 3, it still wouldn't print size? True. True, because, the reason is, it would just, so if we just change this to 3 here, what it would print would be, meow, you know, whoohooo, and then undefined. Because what it's going to do, once it gets to the index 2, so once i is 2, it'll say box at 2, and it will do a lookup at our box, and we don't have a property at 2, right, we only have a 0, we have a 1, and then we have a string size. So it's never going to return size in this loop. And this is a loop we tend to use because again, when we're using an array, we're not usually using it to have numbered, or string properties, that are characters like size, for example. So what, in the console it identifies, and I put size, like you'd have size 9. And then I have 0 in quotes, and 1 in quotes, and when I just hit box, the size part doesn't come out. True, so the console like, just prints what it, it depends which console you're using, and the version of Chrome and all of that, but, for example, I have three undefineds. Let's see, so we have box, oh, that's because, yeah. Look at that, this needs to be i. Cool. So if we look at box, it appears just to have two things. But if you actually if you console.log box, sometimes you can see, let's see. So size is there. It's just not at a numerical index. And your console probably isn't going to pay attention to it because it's assuming that you are only worried about the numerical indices, and it's really up to whoever is designing the console, so what it's showing, what it's not showing, because really under the hood, there's a lot of different methods and properties on everything. Especially natively, that it's just not going to print out. And that same thing with length, like length of 2 and I just hit, not when I do console.log, but. Do you have a question? I just want you to verify that arrays stay ordered and objects don't? So the arrays are only ordered because of the numbers. Because it starts from 0 and it counts up, that's why there's order. There's not, nothing like magical happening, that like is keeping track of the order, it's just the numbers.

Native Properties

So, that second line box, square bracket size = 9. Uh-huh. There's no way to get that value out by using indexing with integers? True-- true because the only way you can get the value 9 out is by looking up, looking it up by the property name, which is size, which is not a number, right? Arrays can be accessed by index number as well as property names? Yup. So here I am using dot notation. box.size, box If you look at box. And then we have our numerical indices as well, we have meow, we also have whoohooo. Is it better to say arrays can have properties and indices? Mmm, I don't, mm, I'm not sure. Is Box an object that has an array as a property? Or is it an array that has a size that--? Box is an object. It's an array, but an array is also an object. So it's both an array and an object. And the thing that's special about arrays is not really the numerical indices, it's really the properties and the methods that come with it. So that .length property is something that makes arrays different, and based on the .length property, you can calculate a lot of things, so as long as the array, so if you have an object that's keeping track of the length, and then you have the methods, the array methods, that are working off of the length, then you have an array. So you can create your own array just by having that magical length property that's keeping track of the length. And, so how the length property works, it's only going to keep track of the highest numerical indices. So it's going to ignore size, it's going ignore itself, like length itself is a property, right? See we have this dot notation box.length? You know it's not counting that, it's not counting box.push, you know, pop, things like that. So it's only going to be keeping track of those numerical indices, and based on the length, and the numerical indices, those methods can do cool things with the data that we have stored in this array. Slash object. So if box is an object, then length would be three. Is that right? Well, objects don't have a length property. Oh. Yeah. So because arrays are objects, let's say you have an empty array and you're assigning meow to the first index. Behind the scenes does it add a property called 0 and then the value called meow or whatever you're assigning to it? Yup, that's a perfect way to think of it. And that's kind of why I draw it like this. You can kind of see, this is kind of how I think about it, that it's just an object and it has these properties on it, and it has the length property and a bunch of methods that give us some powerful functionality, like we can sort things, we can reverse it, slice, all those things that you can't do with an object. Yeah. So when you're console.logging you're actually logging the object? Mm-hmm. So that's why it shows the size. And that's why we could use it for this other for loop, remember, the for in loop will work with, will work with box even if it's an array. And that's because, because it's an object. (quiet question from the audience) Yes. In that case you do because it's just going to be looping through all the property names, that includes numerical properties as well as named string properties, yeah. Oops. Cool, you had a question? There was a question on why you would want to add a property to an array. Um, For different reasons. This will, it depends, you can add methods to it, to the prototype and things, that will add, you can extend functionality that way. You can, I don't know. And then I guess following from that, another question so if I have an object with an attribute called length push pop, etc., it will be an array? Yeah, so if the length property is keeping track of the numerical indices, every time you add something to it, then yes. If we use object notation when you add those properties to it then it's not going to make it an array. You can, you can mimic an array functionality. You have to build that into your-- Well that, that's what he was saying, if you do build it out, they're both objects. Right. One's native, one's not. The only hard part would be keeping track of the numerical, so you'd have to have some way to know whenever you're adding a property, to the object, so that you could adjust the length property. So. You'd have to use a get or a set or something. Adding the length property to that doesn't make it an array, you're not going to pick up automatically pop, push, slice. You'd have to build all that out. You'd have to build all that out, all that functionality. Yeah, right, it's a little more complicated, but. Cool. Sweet. All right, what about this? What's this going to log? So is it even going to work? No. Why? Because length is in quotes. It's looking for that. Because length is in quotes? What is length, so length in quotes again remember, it's the same as dot notation. Right? True? Length in quotes the same as dot notation, thumbs on that? Cool. So the length is 4. Right, and that goes again to like how length is working. So length is keeping track of the last index. It's not keeping track of like the number of, of actual values in the array. They wanted me to ask you, in theory you can get away with never using dot notation and only using bracket notation? True, you can, you could only use bracket notation. It's just more to type. Cool. What about this? What if I change this? Undefined, because length does not have quotes around it. That's tricky. That's not fair. I know, I just, you guys should just know that I'm tricking you the whole time. And look out for that, and think about those rules. And as you're coding you should be actively thinking about the rules. Is, you know, is this, you know, series of characters, is this a string? Is this a variable? Is it an expression, that needs to be evaluated? You should always be thinking that and keeping track of what it is. And then if it is a variable you should also be keeping track of what that variable can represent in different scenarios. All the time. Can't just take it for granted. Can you go back one slide again? Sure. What did you say at the very tail end of this explanation, you said it only looks at the last-- Yeah, it looks at the last or the highest, the highest numeric index, and then it adds one. Because the length is always plus one. Because it's zero-based. Because it's zero-based, absolutely. That's weird. That's programming. (laughter) Yeah, when I tell some of my students, you're my number 0 student. (laughter) Let's see, so we did that one, and then, here, so what if we, what if we, what if we wanted to find the last, we wanted to look at the last index of our array? Minus one? Like this? The length? Just minus one. That only works with slice. You're thinking of slice. Box.length? Box.length-1, totally. So box.length, right, is going to be 4 and then our last index is 3, as we can see here. So we need to subtract 1, and remember back to the thing about the bracket notation being able to evaluate expressions? This is a perfect example of when you'd want to do that. So here we're doing a property lookup, and we're doing math. So all those things are happening, and those are things that you cannot do in dot notation. And it just looks weird, right? Like, look at this. Right, it just doesn't look right.

Arrays Exercise

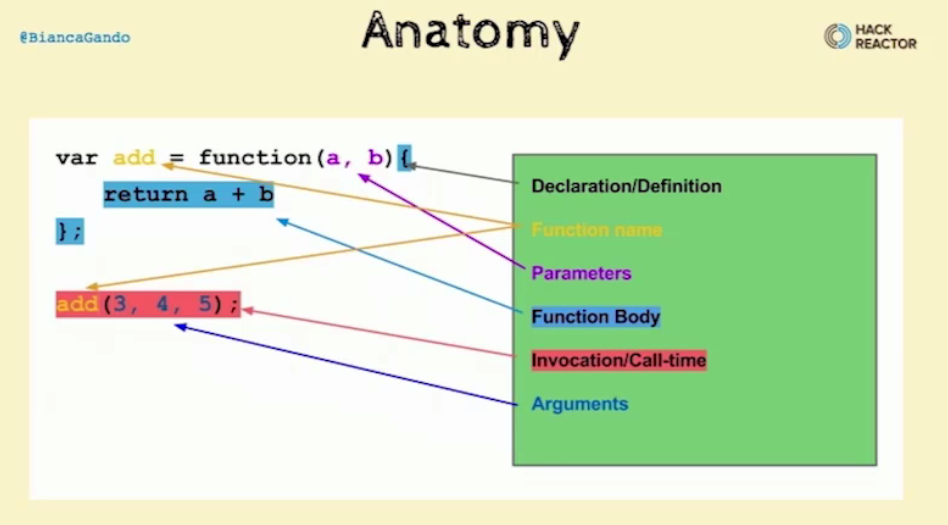
So, actually I'm going to do nesting in a minute. And it's going to be exercise time. So again we're going to jump right to the exercises. We're going to go to arrays. And this is a little bit, a little bit longer, so maybe 30 minutes, let's say 40 minutes, let's do 40 minutes for this one, and we're going to create our second data structure, which is going to be a collection, which is going to be an array of our animal objects. And that's how we're going to be representing our data for our application, for eFarmony.

Arrays Exercise Solution

Let's just quickly go over the exercises from before lunch, so these are our array exercises where we're practicing creating a data structure for our eFarmony application. And this is going to be a collection of our animal users. You could imagine your application, if you have different pages, you'll have perhaps, you know, a user profile page, if you think of your Facebook profile, that would be the model. Our first object from step one, the objects portion, and then there's the portion where maybe you have like a browse, where you can sort of browse the different users on your website or a search function where you would then have a collection of all these different users shown on that page. And so this would be like different scenarios where you'd be using either, just a model data structure, versus your whole collection, which would be an array of user models. Any questions on sort of just that general concept of sort of like, how to architecture data in a way that makes sense for the view that you're, that's going to correspond to the situation that you're going to use it? Okay. Did you have a question? No. I don't think so. Cool. Okay, so I'm just going to quickly go over some of the exercises. So here we're creating a noiseArray, and we're just adding, we're adding a string there called purr. Notice this is just like the object-literal notation, where we can just add it in there. If you wanted to do two, we can just say, I don't know. I don't know, what's a sound? What's a sound? Oink. Oink, Okay, so we have a cat, also oinks. So just like in our object-literal notation, you can see that we have our values separated by commas, and that we just put them directly into the brackets, the corresponding brackets. And that's array-literal notation, same as, or really similar to object-literal notation, insofar as just, you can just automatically add your values in there, you don't have to necessarily use bracket notation to add them, or a method like shift or unshift. I'm sorry, unshift or push. Cool, so and here we are, we're using unshift for our noiseArray. What is this method do? Puts it in the first position? Yeah. So unshift will have your array and it will put that value at the first position, so everything will be shifted over once, and the 0th would then be whatever you pass. And so this is, unshift is what we would call a native array method. So it's a function on the array that comes out of the box with JavaScript. And then push is another one of those array methods, and again a method is just a function, and we can tell that this is a function, right, because we're calling it, with those two parentheses right there? Just like we would call a regular function if you said var x equals function, you know, and you returned some, whatever, and you wanted to call it, you'd just say x parentheses, just like, oh I don't have my, just like this. So, var x = function(){}, right. And if we were to call that function, we would just do that by having two trailing parentheses, right? And if we said var obj equals our object and then we said obj.x, we have our function, uh oh. Maybe it doesn't like that it's empty. Okay. And so just like we can call x by itself like this, right, we can also just call it on the object the same way. So that's all our array methods are. And then here on line 15, we're just adding using bracket notation and a number, we're using, and we're just adding the value growl. And so then we can console.log our noiseArray.length. And what is that going to log? How about, who haven't I talked, how about Keshore? What is this line going to log? Four. Close. It's five, because I added an extra one. Yeah, so this is going to console.log five. And then if you want-- If it's noiseArray Oh you're right. Yup, you're right, it would be four. Do we get a point for that, for? Yep, extra points. Yeah. So many points, bonus points. (laughter) And a gold star. OK, so line 20. Console.log, we're just inspecting our array, and our array is going to look something like this. And, so animal.noises we just add the noiseArray, and what's happening here is we're just overwriting that empty array. So we're not actually adding something to the previous array, we're just completely overwriting it. When we have, when we see this equal sign, that's not augmenting, that's not adding or anything like that, it's just going to completely overwrite it. So that's something important to keep in mind, and now you can see your animal has, you know, its username, tagline, and also this array at the property name noises. Cool, any questions on that so far? No? Okay. So, then here we go, we have our animals collection. Which we're just going to initialize as an empty array. And then what this is going to be is it's going to be an array of animal objects. And so here we have our first animal that we're pushing on, which if you remember is the cat Mittens, and then we're going to take quackers, and we're going to add quackers to where? The end. To the end. So that's actually, it's going to be the length, so the length is the last index, or is the last index plus one, so this would be the same as saying, array.push, for example. You might even imagine that that's what array.push does under the hood. It wouldn't hurt you. And then console.log animals again, just to check to see where you are, and to make sure that you have what you're expecting, and it should print out something like this. So you see we have an array, and then inside of the array we have one object here, and then we have the start of another object, here. But then again we can create an object dog with our, using object-literal notation. And then we can create honeyBadger, and we're going to be using all dot notation here. And also then we can use the push method, which can take more than one argument. And here's where that array.length metaphor breaks down, because you can have more than one argument. And then you can console.log animals.length, and you have four. And that's how you create a collection and, has anyone used Backbone before? A little bit, so this might sound a little bit familiar, I worked with a lot of Backbone. And so this is kind of how you can like envision sort of like the models and collections of Backbone, and also just like general MVC concepts. Cool, any questions about these exercises before I move on to functions? Good, thumbs? Nice. Awesome.

**Functions**

**Anatomy of a Function**

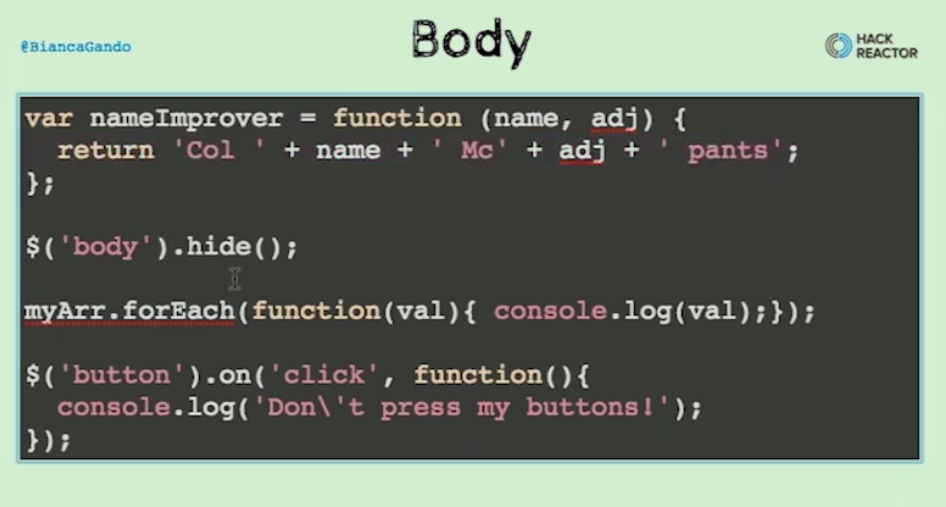


(music) So now we're going to move on to Functions. What do we know about functions? Let's see. Joe. What do you know about functions? Just one thing? Functions are just small modules of code that try to isolate a certain operation. Sure, so you can use functions for encapsulation. What about Tanner? What do you know about functions? This is kind of the part where the code does something. Yeah, yeah! So it's, you could think of it, kind of like a button and you can hit the button and then it goes and it runs something. Uh-huh. Let's see. What about Ben? Functions are Objects. Functions are objects. Nice. Extra points. Let's see, who else wants to go? William, did you want to go? I know that the parameters can be passed in willy nilly. Yeah, so you can pass arguments and there's parameters that hold values. Awesome. So these are all true. So the way that I think of functions are, they're an object, just like Ben said, but they're an object that does something. And the fact that functions are objects in JavaScript is a really interesting concept. The fact that we can assign them to variables and pass them around as data, which we'll talk about more tomorrow when we get into like higher order functions is really, really powerful. And so that's just a cool thing and so we can pass them around and one metaphor that we use a lot at Hack Reactor is we talk about how a function is a blender. And we say, so a blender is an object and you can have a factory that makes blenders. So if you have some constructor that's making more functions, for example, you could do that. And you can give the blender, I can give the blender to Grace and Grace can give the blender to Kashor, for example, or we could set the blender down, put some fruit in it, right? Arguments, and press the button and make juice. And so once you press the button, that's running the function, and the metaphor of the button is those two parentheses. The fancy name for that is the invocation operator, but, two parentheses next to each other. And so that's sort of the metaphor of how I think about a function as data and also as something, a unit that has an action, and that can be run. So the next thing, that I've also heard about the way they described it is it kind of takes care of some detail work that we don't then want to be worried about so it's just a very easy way to do something that then you know it works. Kind of like abstracting it, so I don't have to-- Yeah, yeah. Really worry about well, how does it figure out the cube of X or whatever, it just does it. Yeah, yeah and so yeah, it's, totally, and so when we get into more functional concepts, we talk about functions as units of abstraction and actions that we can then combine in interesting ways. That could be very powerful. Some people think of the properties being the nouns and the functions being the verbs. Yeah, absolutely. I like that. So functions being the verbs and objects being the nouns. Cool. And functions are scoped. Yeah, functions have scope. That's super, super important. So whenever we call a function, it creates a scope. Every time. That's important. Did you have a question, Mark? Yeah, I just tried to figure out if they're asking about nesting in the arrays chapter, did we skip that? We're going to get to that. Okay. Yeah. That's what I was wondering. Cool. Sweet. So we're just going to do an overview of the anatomy of the function. Really quick, and then we're going to go on to nesting. I just kind of wanted to do something new and then we're going to go back and see if we still remember the rules that don't change about, and we're going to apply that to a nested data structures. So here's just a quick anatomy of a function. Some keywords that are important. So we have like the function definition or declaration and that includes things like the keyword function, the parentheses, and the entire function body. And I'm sorry for those of you who are color blind. I put arrows to kind of help out there but I don't know how else to visualize it. And so then we have the function name. You can have a function name, or you can save an anonymous function to a variable. Both ways you can refer to the function later, just like a variable. Then we have the parameters, which are A and B in this case, here. A and B. And then our parameters get values when, during, at call time, they get values when we pass them arguments, here. And then call time is when we have the two parentheses. That's when we press that Go button, and we also call that, so call time, invoking the function. I don't know, that's all I can think of right now. Do we have to, do I have to put var when we define it-- No. You don't? But is that a global scope thing, or? Yeah, it's a hoisting thing. So I recommend that you always just say var function name equals function. Yeah, it has to do with hoisting and scope and it can get a little hairy if you're not careful.

Definition

So now we're going to look over some functions that you may or may not have seen before and kind of like identify the parts of these functions. So we have four of them. Some of them I took from jQuery-- has anyone here done jQuery before? Okay, cool. So a lot of people have done, done jQuery. So the first thing is the function definition. So where do we see some functions defined? So let's just go in a line. Let's start with the front, this way. At the top, there's an anonymous function being assigned the nameImprover. Yeah, so we're defining a function here. And then, Rich? We're looking for where we're assigning functions? Where we're defining functions. Defining functions? It could be assigned in a variable. It could be anonymous. Would that be the body hide? Nope. Down there. Close. No clue. Right there, that's your next one. We're defining a function there. It has no name, and that's okay. That one doesn't have a name? Yep. There's no name. Oh, it's just passing val? Mm-hmm. So it's a function definition. It's also an argument. Yeah. And who sees the last one? Tanner? There's a function on click there. Right here. Function definition. So these are the function definitions in this code. Oh, and just for clarification, nameImprover function is this handy function that we use. At Girl Develop It for when our friends have babies or they have names that need to be improved. Body.hide is a jQuery thing that's just going to hide the body. So if I just did you'll see here. Wait, is that even jQuery? Yes. Oh, no there's not jQuery. Let's see if it works. Oh, cool, so it does work. So body.hide will just hide it and then .show will it and then array.foreach, foreach is a native, like, iteration method. It's going to just loop over the array and call this function a bunch of times and then button on click is another jQuery jQuery functionality and it's just adding an event listener so whenever you click on the button it's going to console log this. This "Don't press my buttons". So that's just an explanation of what functions we're looking at right now. Cool, and so what about function names? And I use name a little bit loosely. I refer to it both as like the function, the name function as well as like a variable that repre, that has a function stored at. So let's talk about where do we see function names? So can we start with just Grace and we'll just keep going? So you can anticipate when you have to go next. I don't see a name. For function? Yeah, so like, here for example would be a name. Oh, okay. That's the name for the function Yeah. versus the variable. Yeah, well it's a variable. I'm using name here loosely to mean, like, both the variable name that it's stored at and the name itself, yeah. 'Cause there's another way, like, we could write function like this function nameImprover, name stay same. So this is another one. So here's, this is just a really complicated algorithm that will just keep your name the same versus this other, this other one that will change your name and make it better. So this is another way to name our function. Is that the same as is that the same as the first declaration of a function, where, in the sense that you're adding var in front of it, versus adding it to the global? I don't know if that makes sense. So when we name-- Do you never want to do that one that you just did? The same stay name or whatever? I don't know. You could do it this way. The only thing about doing it this way is that your function name and definition is hoisted and that can be, like, if you, it can, like, get messy. Yeah. So it can get messy. So I just recommend that you always just use this one, 'cause you just, you know that the rules for variable hoisting and things are the same, you don't have to keep track of this and, Okay. Yeah. There are some situations, like if you have two functions are the same name in the if else statement, and then you run it after the if else statement, then it can get mixed up and do things that you don't expect because of the way hoisting works in this case. Okay. So you can do it, but you just have to be careful. But just to be on the safe side, just always do it this way. It's not going to hurt you. It's just var, just use one extra word, and it's not going to make a big difference. So, okay, so back to where we were. Here is a name, then we also have a name here. And then William, see any more names? Well you just added one called nameStaySame. Mm-hmm, what about, any other ones? I don't see any other ones. What about hide? That's a name, right? Yeah, yes. Cool. Kim, do you see any name? On. On, absolutely. On is a name. And Andy? Foreach. Yep, foreach is a function name. Joe? Log. Yeah, log. See, are we missing any? Oh, there's one more. It's kind of a trick question, though. Anyone see it? Dollar. Yeah, dollar sign. So this jQuery dollar sign is actually a function. It's not a special syntax. It's just a function. So you can imagine that jQuery, somewhere in its library says var dollar sign equals a function and then, you know, it, like, returns like a jQuery object somewhere, right? Oops. More like this. So, some things that may seem a little different are actually following the same rules and I find that very comforting actually. To know that. That jQuery isn't doing something like totally crazy that I don't understand.

Body

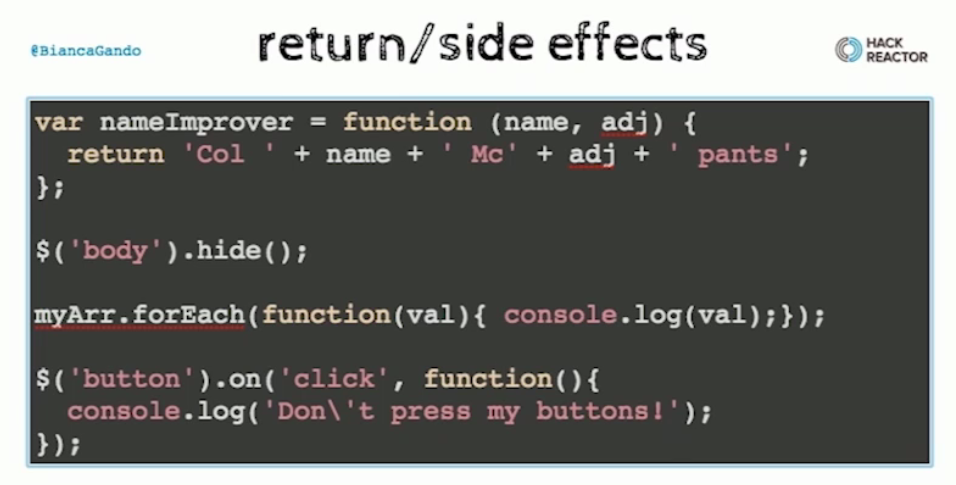


(above is used as a sample to find all the instances of things like names, body, etc.)

Now what about function bodies? It's a little bit different than a function definition just because the body is what's located between these two parentheses. I'm sorry, these two curly brackets. And the important thing, while you guys are searching for more function bodies, the important thing to note about a function body is that it's never run until call time. So the code located within those curly brackets is not interpreted until you call the function so in this case, for nameImprover, since we're never calling it, this code here, never runs. And it has no real value. Cool. So let's see, who is next? Is the body before the .hide? The body? Not quite. So we're looking for two curly brackets. That's how we know it's a function body. Unless it's an object or an if statement, but on this slide. How about after function val? Yeah. So right here. We have a function body. And then Ben? In the on function, the second parameter is the, yeah, and then starting at the. Yeah, so right here. That's a function body. And Jake, what's important to remember about function bodies? They they are the verbs. Functions are verbs, that's true. You can, what's it called? Phone a friend. (laughter) Oh, I can keep 'em coming. Kim, help me out. My lifeline. What was the question? What's important to remember about function bodies? They're inside brackets. Yes, that's true, but there's something else. I'm looking for like one specific thing, anyone remember? They're not called 'til-- Doesn't call-- Yes, so it's not run until we call the function. Awesome. Cool. Speaking of calling the function. We have this thing called call time invocation so just for fun we can like we can improve someone's name. Oops. Uh oh. Okay. So I'm just going to put this function here and notice, I just put nameImprover there. We have it. Right, when I type out nameImprover it shows me what the function looks like. That's not call time. That's just me checking what is stored at that variable, just like when we say var X equals two and I just type X, it returns two. Same thing. So How many people here really love their name? No? Everyone? Okay so then we have three people who really love their name, so everyone else could use some name improvement. So everyone else just volunteered. (laughter) Full of tricks and traps. I mean, it's Halloween time, you know, it's time for those things. Okay so of those people, all right, you can recommend a friend. Who wants to nominate someone? Tanner. Tanner, okay. Thanks. (laughter) And who has an adjective that I can borrow? It has to be a nice one. It can't be like stinky or something. Buff. What? Buff. Buff? Okay. Buff. Nice. Awesome. So Colonel Tanner McBuff Pants. Nice. You can call up your lawyer and arrange a new name if you want. I know it's way better, right? I'll get right on it. Yeah. (laughter) All right, I'm going to call you that for the rest of the day. I appreciate that. Yeah, no problem. So notice here, so we have let's just look at the inside of our function. That we have our parameter here, name, and our adjective here, and they have no value until we pass these parameters, Tanner and Buff, right? And then suddenly they have value and suddenly the code in this line, in between the, inside the function body, gets called. And so that timing, that order, is really important. Cool. So Where else do we see call time? Hide. Yeah, hide, there is a perfect example. So we're calling hide. Where else? So where did we leave off? So we had Jake and then Kevin. Where else do we see call time? Foreach. Yeah, right here, absolutely. And then, Kashor? On. Where? On. Yeah, on is call time. Ah, let's see, so I guess we'll start again at John. Dollar sign. Yeah, so here we're calling jQuery. Anything else? Oh there's one more. There's log and val. Yeah there's just log. So console.log. There's two of them, so. Are you saying call time is another word for invocation? Yeah, exactly the same. But not on val because that's just a function? Val here? Val-- That's a parameter. Parameter. That's just-- Oh that's a parameter name. But that, that parentheses-- This one? No. That's defining the function. Oh. And the parameters are going to be passed into it. Whenever you see the keyword function followed by two parentheses, that's not call time. That's part of the function definition. Got it. So the main thing to pay attention to is just the keyword function that's followed by parentheses, then you know that that that's just, that's going to be the parameters. So like function val, that's like kind of in line definition, and? Yeah, so that's just an anonymous function that we're passing to foreach and then val is going to be a parameter to that function. Mm-hmm. Cool.

**Arguments and Parameters**

So now I want to talk a little bit about the difference between arguments and parameters and this is something that's like, really, is really confused by people. So they're, they are different. They're not interchangeable. Invocation and call time mean the same thing. Arguments and parameters are fundamentally different, so parameters, like I was saying, are right after the function definition, or the function keyword and they're just placeholders. So they're like variables, right? They hold values, but before call time, they have no value. They are undefined. And the way they get a value is if you pass them arguments, just like when we were improving Tanner's name. What was it? Buff. So, name and adj, at this, when I'm walking through the code, at this line, the first line here, name and adj have no value. This isn't even run yet. So it skips to the end and then once it gets here, that's when we're giving name and adj, adjective, the value. So Tanner and Buff are then given a value to our parameters and order is important, so they map by order. So name is the first one. So the first argument is going to give the first parameter its name, so name is then Tanner and then the second one, Buff, is then adj. And there's no way you can, you can't mix up the order. The order is super, super important and that's the only way that your code knows which argument goes to which parameter. Cool. Which one, so name and adj are the parameters? Yes. And Tanner and Buff are the arguments? Yes. Okay. And the words themselves aren't that important. It's just the concept that these are variable-like and that they're placeholders, and this is the actual value, right? And you can kind of tell that these are variable-like just 'cause they don't have, like, they're not keywords like function or return, but they're also, they don't have quotes around them. They're not any other kind of data type. So where do we see, let's go through parameters. So we just said that name and adj are parameters. Where else do we see parameters? Where did we leave off? Did we leave off at Tanner? Okay I'll go first. So parameters are just the placeholders. All right. So after, after nameImprover. Yeah. Those are arguments, okay. You can phone a friend if you want. How about after foreach? Close, so foreach is actually a function that's being called. All right. But here, that's a parameter, val. That's a parameter. And I think that's the only one 'cause this function has no parameters. And then what about arguments? Where are we passing arguments? So arguments are passed to the function at call time. So, Grace, I think it's your turn. Would that something like a on click to the console? Yeah, yeah. Click. So click is an argument. It's just a string. And this function is also an argument. Yeah I was wondering if the foreach isn't the function parameter or am I mixing up? Is that the argument? So this is the argument right here. Is the entire thing the argument? That's the argument. That function is an argument. And where else do we see arguments? My turn? Mm-hmm. The jQuery. Yeah, so. By button. Oh yeah, here. So this is also an argument. Do you see any more, Kim? Don't press my buttons. Yeah, don't press my buttons. So console.log in the console.log. Cool. Any questions about that? It's kind of a, it's kind of a gray area. It's easy to get them confused. Any questions of clarification before I move on? Cool. So the next thing I want to talk about is what it means to return a value versus having side effect.



So if you don't have an explicit return statement in your function, it's just going to return undefined. Okay? So if we want a function to return a value, we have to explicitly say return and the thing about return is that it's going to immediately break out of the function and return that value, whether if it's in an if block, or a for loop, it's just going to immediately return out of that function. So you have to be careful about where you place your return statements, but if you do in fact want your function to return a value, the only way you can do that is by having a return. And everything else that you do that's doing work is a side effect. So this console.log here, it's considered a side effect, 'cause it's not actually, this function is a not actually returning anything. It's just console logging. So if you change the assignment on a variable, but don't return anything, it just, that's a side effect of-- Totally. Exactly. Or if you're selecting, if you're creating dom nodes and appending them inside of a function, that would be considered a side effect and not returning, so you have an explicit return statement in which you, like, return, like, some data, right? Or you have side effects. And those are the two different things. Cool. Cool, so just a quick review. Where did we leave off? I think it's your turn, Andy, right? So what is, what is this add thing, this add function going to return? This is testing your math skills. Seven. Hope you studied. Yeah, seven. Awesome. And take a moment and explain to the person next to you how you would change this add function so that it would add to thirteen. And you can't minus one. Here, let me change this. Okay. Actually it doesn't matter. Without doing ma, without, someone, I did this the other day, and they're like, "Yeah, I just, A plus B minus one." (laughter) That's not what I'm asking. So, yeah, explain to your partner how we would get this to return thirteen without doing any extra math. Just changing-- Just change the function? Change the function. So explain to someone next to you. Or in the chat you can, you can type it out if you want. (class talking) So what we're changing is the function itself, not the arguments. What'd you come up with? Anyone come up with any brilliant solutions? To this complicated coding problem? I would add comma c so that the definition has a comma b comma c. Mm-hmm. And then I would return a plus, or b plus c. Totally. And here we see that we don't need to use all the arguments, or recon, or even do any work on a param, on parameters. We can just skip them and JavaScript's really flexible like that. It doesn't mind and even when we did only have two here, a and b, notice when we passed three arguments, it just ignored them and that's one of the cool things about Java, it's one of the dangerous things about JavaScript, that flexibility, but it's also one of the cool things. And what comes from that, or a cool thing that we have in JavaScript that works with that is this thing called the arguments keyword. And arguments keyword is a special keyword in JavaScript that lives inside functions and it gets a value of the arguments that you pass in in the form of an array-like object. And what it would look like if we pass three ten and five to our add function and then we just console logged the arguments, it would log something that looks a lot like an array. It looks like an array of all the arguments in the order that you pass them and so this is really powerful when you don't know how many arguments your function can take, so you can have that flexibility. Like maybe you're going to add several numbers, and you don't, you want to have it flexible. You can add a bunch of them, so you can loop through it and just add it that way. Another cool thing that you can do with the arguments keyword is you can you can have different versions of functions. So you can say like if you pass three arguments, do this. If you pass five arguments, you know, do this. So you can have different options for how your function works. So if we define the function with the empty brackets instead of a comma b, it's just empty, but when you call the function, you add, you want to say three, four arguments or param, arguments. Then to access those you have to kind of loop through this argument's list or array, I mean. Yeah, so if you had a bunch-- Because that's how, the only way you, like, reference them. Do I, I might have that slide. No, I don't. So if you did want to have this all add together, you know, what you could do is, you know, do our for loop, remember this? So arguments is not an array, but it does have a .length property, which is handy. And then we can have, like, results equals zero, and then we can say results plus equals arguments at i, and so this is how we would then just like, if you wanted to add all these arguments hopefully I have no syntax errors, let's see. Cool, so 138. So that's one use case. Cool, so how then would we, if we wanted to have it equal 18 using the arguments keyword, what would we change in the add function? So if we wanted to add something here. Arguments two? Yep, so it'd be arguments bracket two and that would return 18. Cool. Awesome. And so I said, it really isn't an array. It's an array-like object. And what does that mean? Why is that important to us? It's important mostly for when you are trying to use array methods on it. It won't have slice, for example. Which is a little unfortunate, but there are methods that you can, there are ways that you can turn it into an array and that, it's kind of a little bit out of the scope of what we're doing because it goes into the prototype but something like, I could even be typing this wrong, it's like prototype.call .slice.call something like this, so you can just Google it actually, the exact syntax I can't remember. I pasted it in chat. Is it? Okay, cool, arguments zero. Thank you. So this would be an example of how you would turn it into a, into an array. And it's array. It's array.prototype. Mmm, thank you. Awesome. So if you want to use any of the handy array methods, you will first have to turn it into a real array and then you can do those operations but if you're trying to do it without it, it's going to give you an error. Cool. Oh, also, functions are objects. Which means those rules that we were talking about before with the dot notation, the bracket notation, all remain the same. I'm not going to go into a whole spiel about it. But I hope by now that you trust me. I know how to show you all the examples that dot notation and bracket notation work the same on functions and you actually use this a lot for when you're making constructors.

Constructors

I'm just going to jump right in to looping and, well, I'm sorry, that's a misnamed slide. We're going to talk a little about constructors and we're going to be using loops and we're going to build on this more tomorrow when we get into the functional programming so we're going to do some things here, just the manual way, which tomorrow we'll sort of apply some functional approaches using the underscore library. But the first thing I want to talk about is the idea of a function as a constructor. And as we see, we have this function called AnimalMaker on the screen. We have the capital A to let other programmers know that this is a constructor and what a constructor is, at its core, is a function that returns an object, that's it. So we see here we have our AnimalMaker, it's just returning a function, I'm sorry, it's returning an object, and what, what does this object, what properties does this object contain? Speak. Speak. Yeah, so speak and it's a function that just console logs the name of the animal. True? So then if we wanted to, you know, create an animal object, so var my animal, oops, equals, just a second. We had to put our function in there so var my animal equals AnimalMaker and then we could pass it a name. Who has a pet that they love? What's it's name? Cheetah. Cheetah? Cheetah or Cheetoh? Cheetah. Cheetah. Okay, like that? She look like a cheetah. Oh, cool. It's a kitty. Is that how you spell Cheetah? Yeah, okay? So now, we just used our constructor function to create our animal object. And here we are. And then, if we wanted to then, if we then wanted our animal to to speak, how would we go about doing that? Yeah we could use a dot. And then what? Yep, good job. .speak? Like that? And what's this going to return? Function. Or no, there is a return, isn't there? What's going to happen? Don't we have to put the parens in there? Yeah, absolutely. So if we don't put the parentheses, we just get a function. And that's not really what we expected, right? We actually, we want it to do what we expect it to do, which is to console.log my name is name speak and so it says, "My name is Cheetah". Cool. So that's the essence of a constructor function and so we're taking some information and we're turning it into an object that we then ha, and then we have a model of our data that we can use in some way. And notice the rules again, they don't change. The ways that we access our methods are the same when you can use dot notation, or we can also use our bracket notation. Like this guys. Like this? What am I missing? Quotes. Quotes, exactly. There we are. So and then, I was talking with a student the other day about-- So in bra, I'm sorry, in bracket notation you have the invocation also? Yeah. Okay. Absolutely. Okay. That's important. Mm-hmm. So So I was talking with someone the other day about when you inspect when you inspect the inside of a function here, we don't actual, it's not saying Cheetah, it's saying name. And we got into a discussion about why why doesn't it just say Cheetah, if it's going to be Cheetah? And the answer is is that before you call a function, your parameters don't have a value, remember? And so even though we know that it's going to be Cheetah, that that variable gets its value dynamically when you call the function, and so that's something to note when you're, when you're going through your code.

Looping

Cool, so we have our animal maker, it's returning an object, has speak, like we can even put like a name function on it if we wanted to, just like, so we can remember who's who. We say name, his name and then we can like I don't know owner, maybe. So what if we wanted to loop through these animal names and create an animal object with each one? How would we go about doing that? Actually, I'll give you a moment. I'll give you like five minutes to try this out on your own. So what we're going to do is we're going to loop through for each name, we're going to pass it to AnimalMaker and create a animal object. Cool. Let me know when you guys have done that. And then to the question about the console log, with the plus sign, so you can use a plus sign if you want. What it does, though, to this variable is that it forces it into a string, so if name was an object that would run into some problems 'cause if you stringify an object, it's like saying this, string plus some object, it's going to quote object object and when you're console logging, maybe that's not going to enlighten you the way that you need to and if you wanted to, to print out what it is, you can just use the comma and it will just print out the object prettier. Oh, I forgot to mention that you want to add it to your farm. So you want your animal objects to be in your farm, which is just an in, it's just an array. So when you're looping through your array, you're creating your animal objects and then adding them to your farm array. All right, so, what's the first thing I need to do? I suddenly, guys, I have this like, periodic amnesia that happens, where I just forget everything I ever know about Javascript, so I'm going to have to rely on you. I'm kind of in a high-pressure situation, I'm on camera, so I need your help. Really desperately need your help. How do I, how do I make my farm? You guys are not, you guys are not the rescue type, huh? I'm here, I'm on the spot. I'm sorry, what's the question? So how are we going to loop through this array-- For, foreach. All right, we're going to make a for loop. var key in, should I do it this way? Yep. Yeah? Probably not, right? We kind of talked about that. This is for objects. So we want the i one, right? animalnames.length, right? I sometimes make mistakes, though. Let me know. Okay so I wrote my for loop, so this loop is going to loop through all of my animal names. And now I want to create an animal object for each one. Call AnimalMaker and pass in the index? Pass in the index? i? As names variable. Think it's in the reasonable syntax? Like that? Brackets. Bracket. And then what? So now we're creating these, we're creating the animal objects, but they are just going out in space. We want to add them to our farm. Otherwise, they're wild animals and this would be the jungle. Farm square bracket i equals AnimalMaker Or from that AnimalMaker? Oh I see. Whoops. So is this going to work? Yes, oh I see. I think it'll work. It'll work, that's one way to do it. Awesome. So this is how we're going to create our farm. We're going to use our constructor function. We're going to loop through our animal names and we're coming up with these robust animal objects that have methods on them, so all of our, all of our animals in our farm can now speak and if we later wanted to change our constructor function a little differently, we could maybe even add more. So here's an example of the answers of how we could do that. We could do farm.push, so the one that we did before was farm at i equals AnimalMaker. And then the second, another way we can do it, we can just use the .push method, where we say var animal equals, again, AnimalMaker, animalNames i, and then we just push it, each one, or we could just do it all in one line. So we could say farm.push AnimalMaker, parentheses, animalNames bracket i. Thumbs on how we got there. Thumbs, thumbs, thumbs. Thumbs in the front. Thumbs in the front, cool. Is there a reason why you're not using new? New keyword to create the? Because that would be, then I would have to explain "this". I would have to explain keyword new. And, at the end of the day, when you use the keyword new, all it's doing is returning an object. So at it's core, like, there's a bunch of different class patterns that you can, that you can use in JavaScript, 'cause we don't really have formal classes, so you can just write a function that returns an object and there you are, you have a class, or a constructor and then there is a way using the keyword new you can look it up, it's called pseudo-classical, that's the pattern and that's, that tends to be the typical pattern, using the keyword new, you see this in there a lot, the .prototype, et cetera. But that's like, I have another class on object-oriented JavaScript. It's a week long, so it's just like a whole nother animal, and I try not to introduce, like, too much extra too much extra stuff. But, good question. Any other questions? Kate's asking how the object is being created. She doesn't understand that. Sure. So we're actually, so this return statement here, these two, these curly brackets here are the object that's being created. So if we just copy this quickly into our console, let me get a little bigger. And we just say var obj equals AnimalMaker so whatever, so how this is working is that whatever this function is returning is then going to be stored in this variable. And so obj, since AnimalMaker, when we know what AnimalMaker is returning, if we go back up, we look into the function body, again the function body is between the curly brackets, and we look at that return statement and that return statement is going to explicitly tell us what we're returning. And notice we have these curly brackets, it's a little bit confusing because there's a lot of curly brackets in there, but they are just, there's no function keyword, there's no if, there's no for, they're just naked curly brackets, and that tells us that that's an object literal, and so that return statement is just explicitly returning an object. So like if we, for example, went back here and we just deleted the speak function, which is adding some extra curly brackets, and we just looked at it like this, you can see that it's just returning an object and that's where the object's coming from. Wow. Mm-hmm. So there's no, there's nothing fancy special here. It's literally just returning an object literal. Can you console log something there? I'm trying to, I can't understand what's in the array. Oh, in the farm? Yeah. Sure. I don't know. Do, do, do. So here we have our farm which is an array of three objects. So for every animal name, we have an animal object for them that we had put into the farm, so if we look at the object, it's just an object that has a speak method on it. So each one will do that and then does that answer your question? How do I get the names to actually show up? Mmm. Well what we could do is we could change this to and then I think, let's see. Let's see if it'll show it now. Yeah. So now you can see the actual name. Okay, let me see what you did then. All I did is I added another property on there called name, just so you could explicitly see the name. Before we just had a function in there. Cool? Awesome. So now what if we wanted to then loop through our farm now and have all of our animals say their name? We're having like, like a farm role call. How would we do that? In the for loop, say animal.speak-- So we want to loop through the farm, right? So it would be a new for loop, right? Yep. I'm going to delete this old one, is that okay? Yeah. Just for room. Whoops, for var i equals zero, i is less than farm.length, i plus plus, da, da, da, da, okay, and then what do we have to do as the next step? Farm square bracket i in there. Mmm-hmm, and then-- .speak paren. Cool. Thumbs on what, where we're at. Can I see everyone's thumbs just so I can have a check? Thumbs, thumbs, okay. Cool. So let's test it out. Oh wait, let's just do this one. Cool. Can you go back to that slide, just, on that function AnimalMaker, that's where we're defining it. We don't need to var there, anything like that? I'm getting confused with the var. Yeah, so that's the other, remember I was saying there's two ways that you can name a function? This was the other way. This is the way I actually don't recommend, but because of slide real estate, I use them a lot in my slides, but in code, you should just use var X equals function, so you could say, I guess this one has more room, you could say var AnimalMaker, oops, not Mater, that's a different thing. (laughter) Actually maybe that's accurate. So it's more or less the same. It does have different implications, as far as like, hoisting and scope things, but, but for today we'll just go with it's the same. We talk about scope tomorrow, so, we can talk about that. Also stack creates a lot of errors too.

Functions Exercise

So now it's exercise time. So if we just go back to where we were, sort of like our table of contents, you can just go to the functions exercise and get started. And again, it's the same, it's all in script.js file in your JS bin and you're going to go through and put some functionality in there. So, and we'll have about, 30, let's see, what time is it? Two? Yeah, we'll have about 40 minutes for these is a little bit longer. So let's go with 40 minutes. There's a question in the chapter in here. Oh. So Pete asked, how does AnimalMaker know when it is passed an array, it is passed an array, and has to iterate through it, versus just grabbing the first value? So we're actually explicitly, where are my slides? We're actually explicitly looping through it. So AnimalMaker, the function itself, knows nothing of the array. What we're actually doing is we're just looping and here, actually this slide up here makes it a little more clear. So we're just looping, so this for loop is actually what's doing the looping part. And inside of the for loop, we're actually passing the animalName, we can see it here. This is where we're calling AnimalMaker, and we're not calling AnimalMaker with the entire array. We're actually calling it with one value in the array, so animalNames at i, so the first one, the zero with index, would be sheep, so it's saying AnimalMaker's, and then you could replace that right here, like, with sheep, for example. So that's how I know, so does that answer your questions? Cool.

Functions Exercise Solution

And it's supposed to be a little bit of a difficult challenge. It's really combining all the different things that we've gone over today: functions, arrays, objects, loops, and just really tying everything together. So, let's see, let me find my solutions. Here we go. Is there a solution file somewhere? There is a solution slide. I'll show, I'll tell you how to get to the solutions later. I don't like to give it away ahead of time because yeah, because then you'll cheat and that would be too easy. So the first thing is for questions one and two combined we have this AnimalTestUser. We, for the first one, though, let me just type out the first version. It's just a, so the first one, we could say var AnimalTest equals a function, and this function takes the username as a parameter and then it just returns an object. So you see that, see how I expanded that object? So there's an object there and then I just expand it out and in there I'm going to write some properties. And the property was, I believe, so it was just username and then username. So this is a little bit confusing. So we have username on one side of the colon, and then we have username on the other side of the colon and username is a parameter. Which one actually, is going to be the one that's going to be turned into the username? So if I called this and I said var like myCow equals AnimalTest and then I passed them, what's a good cow name? Like Bessy or something. Elsie. Bessy? Elsie. I don't know how to spell Bessy, but that's as good as it's going to get. So, which one wins? So is the property name Bessy? Or is that value going to be Bessy? Absolutely, so the value in this case is going to be Bessy. Here's the parameter. When we have the property name on the left side of the colon like that, that's going to be assumed as a string in object literal notation and it won't ever be evaluated as an expression. So before I was talking about how we could use bracket notation to put an expression for the property name? We can't do that with object literal notation. So keep that in mind, that this will always be assumed as a string. If it makes you feel better, you can just put it in quotes to remind yourself that, okay this one on the left is a string and the value on the right is the one that's going to be my parameter, which, once I pass, once I pass the string Bessy, then that parameter has the value Bessy. And if we, if we just wanted to, like, console.log myCow, that username, that is going to print out Bessy, right? Thumbs on that? Thumbs on that, how many people finished that one? Awesome, good job guys! So that's step one. And then the second part of the question was if the arguments are, let's see, if the length of the arguments is greater than one, put the other arguments as a property, or as a an array in, as an array, I'm sorry, let me re-word that. As a property with the value of an array of those arguments. Are you guys following? I kind of lost myself in that explanation. So, other args is going to be an array of the arguments passed in, if there's more than one argument. And so there's a few ways you could have done that. So one way you could do it, so here we have args, which is going to be the length of the, I think a better one would be like arg length, just to be, like, a little more clear on what args is. So we say argLength, so that's going to be the length of the args, it's going to be a number, and then we have-- ArgLength in the if statement? No? Oh, thank you. Someone's paying attention, awesome. So if, so if the length is greater than one, then we're going to enter into this if block and the simplest way, I'm using the concepts that we actually covered in the class, it's just a loop through that arguments array and push them, except for the first one. So that's why our i there, our var i, is going to start at one instead of zero and so you loop through and you push each argument into an array and that would create an array. The other one is what I mentioned briefly earlier, the array.prototype.slice.call will also turn it into an array. But, we can stick to this one because it's more straightforward. The other one goes into prototype land, which is not somewhere I really want to go right now. Any questions about this method? I didn't see that slice.call on the documentation, so call is a function of slice, which is a method-- Call is a method on all functions. And if we have time tomorrow, I have some slides that go over call and apply, if we have time. Call and apply. So, fingers cross. Yeah, so call and apply are two function methods that are commonly used and they're really useful for if we're doing, for functional programming, when you're doing some generic things with functions. But we'll see, we'll see if we get there tomorrow. So they're kind of already built in, not-- Yeah. Yeah, exactly. So like how push is a native array method, call and apply are native function methods. Okay. And that goes back to how functions are also objects, they also have methods, et cetera. Cool. Did you have a question? Nah. (laughter) Cool. Sweet. Da da da. Let's see, let's put on word wrap here so we can see everything. So here's a more complicated version of our first animal user. So it takes username, it takes a species, it takes a tagline, it takes noises. So starting to sound familiar from our other exercises. So what we're doing here is we're creating a constructor function for our animal model. And so here we have an object, and we're just storing it in a variable, which you could do. And again, you could just put quotes around this to sort of, so you know which one's which. There you go. And we're just initializing these values and here at the very end, we are returning that object. And another way that you can, you could do this is simply by just returning the object literal instead of the variable name, but they do the exact same thing. There's no, there's no difference there. Any questions on this one? No? Cool. So here's just an example of how you would use it. So we're passing in a series of arguments and then you can console log sheep and you'll see that sheep is going to be hopefully whoops is going to be an object. So here is sheep, our sheep, and we see it has, its name is Cloud, it's a sheep, tagline is "You can count on me!" and it has a noises array of the noises and it has an empty friends array. Whoops, I lost where I was. Questions on that? Cool. So I'm going to try to go through this quick so that we can keep going. So here's our function addFriend. It takes an animal and it takes a friend, which you want to add to the friend, so if you wanted to simply push, if you wanted to push the whole object it would be animal.friends.push and you would push that friend and so that would be the entire object and so at this point, if we looked at it, we could do this, oops, I keep trying to open the console up in Sublime and it's not liking that. Do we have sheep here still? No, okay. So let's do this uh oh. Da da da. Okay. Okay. So we have sheep and now I think we'll just add Cloud's going to be his own friend, so we're going to give sheep a sheep and so now we can see that sheep has itself as a friend. And you can see there. Cool? And then if you wanted to save space, you didn't want to pass around whole objects, you could just, or you didn't want to save whole objects, you can just keep the username there, or their userid or something like that. And do it that way. And in that case it would just save the name. So here's some other ones. Cow, sheep, and llamas. So here's one way that we can just create our farm by just using our array little notation, pass, putting those variables in there that represent our animal objects like that-- So those were already created in the AnimalCreator? Yeah up here, you can kind of see-- Got it. Here's llama, here's cow. And here's how you can add matches so okay let's see. So we're looping through farm, we could actually use, which might be better, is we could just use a regular for loop with semicolons rather than this for in loop, because again, if we had some properties on our array it might do some silly things that we aren't expecting. But you could also do that as well. So we have farm and then inside farm we have animal for each animal in the farm .matches, so the matches is creating a matches property there and initializing it to an empty array, so this function just gives our, each animal in our farm the ability to have a match. Whereas before, they can only have friends. Like maybe you could only have fr, you could only get matches if you're a premium user. So you can only have friends, and then if you want to do the dating matching part, or something, then you can upgrade them and give them matches. Just don't tell them that they could just put this function in their console and they wouldn't have to pay anything. Not business model-ly perfect. So and then here is a function that gives matches and it looks like it is just going to loop through all those animals in the farm and it's going to push the first friend of each animal as a match and you could get fancy with this and like make it a random match or perhaps in future iterations of your data structure you'll have some some ratings based on a test that they take, and then they'll find, in the farm, other animals that have a similar, a close rating to them and then match that way. Mm-hmm? So when this one animal is it like cow, sheep, llama, is that what it's, it's not like going to the username and matching username-- Right. It's-- It's saying sheep to sheep and cow to cow-- Yes, exactly. So it's looping through, and again, this would this would be better as a for loop with semicolons. It will work this way, but in general you're going to want to use those for loop with semicolons when you're looping through an array. I used foreach. Oh yeah, or you can use foreach. We'll talk about that more tomorrow. Cool and this is how you would use giveMatches, you just pass the whole farm to it and then you can check it by just inspecting your farm and seeing if there's animals, other animals, in your matches. I think I said that all without breathing. Any questions? No? I have a question. Sure. Line 75 and line 76. Mm-hmm? So farm on line 75, is an array? Mm-hmm. Same thing. As this. And then that variable animal is some indexer into the array? What is it actually doing on line 76 where it says farm brackets-- So animal is actually going to be a, is going to be the i, so you could maybe it would look better if it was an i. But in general you're going to want to use the for loop with semicolons here, where you say for var i equals, you know, less than array.length. That one would be better. This will work, but since farm is an array and in a lot of NBC's, especially, or at least Backbone, which is the one I use mostly, is that your collection is actually going to be an object. It's not usually an array. It's usually, it's an object that contains an array, so, so that's probably where the solution is kind of basing that on. Cool. All right, so if we don't have any questions on that, we'll just move on to nesting and that's going to be our last topic for today. Can you, can you just speak to the, the assignment, or the exercise file where you have, under each exercise number, you have like a code area and then you have some items commented out. Is that like a hint for what you're trying to do? Let's see. This here? Yeah, so-- Yeah, so that's saying that this console log will print out this. And what about three below that? Yeah, so we're console logging sheep-- Oh, okay. And so sheep should be an object that looks like this one. So that's what the log would print for that? Yeah. So it's just kind of giving you some guidance into what you should be expecting your solution to look like. All right, thanks. Can you go down to that number eight? Mm-hmm. And so that matches Zanny so that, Zanny is a sheep? And that's why sh, and that's why Zanny is in the matches? It really depends on how you implement your matches. Your giveMatches function. We don't, we don't do species discrimination on eFarmony, so, it's usually just the first, the first one. So the solution does the first one, so-- Oh. So actually, in the solution it would be Moo because that would be the first index. And this one, maybe, is always the last one. It just depends on how you implement it. There's no robust-- Oh, so we're like matching one thing to another. We're just saying the first or the last, or whatever-- Yeah. Oh okay. Yeah, so someone from your friends list will be one of your match. Got it, okay. Yeah, yeah. Right on. And if you want to build that out into something like more interesting, you're welcome to do that. Got it. Yeah. Like matching sheep to sheep, or? Yeah.

Nesting

Nesting Objects

(music) So we kind of play with this a little bit, but I want to talk more in depth about dealing with nested data structures and how to access them, and how to think about them in a way that makes it easier. You're going to deal a lot with nested data structures, usually in the form of a JSON, which is how you're probably going to get your data from your back-end or your API. And so here's just an example of how we get started. So we start with a box, our box, that we love, that we've learned to love over the day. And in the box we can have an innerBox, and so innerBox is a property in box, that contains a value which is also an object. And so this is a nested object. And the rules are the same here. So we can also use bracket notation if we wanted to, and our result is the same. And then if we wanted to add a property to that innerBox, simply what we'd do is we'd continue the pattern. So first we access box, right, so we access the whole box, right, which would be this whole object. And then we access the property value, right so, box, innerBox is going to evaluate to a value, which is the value at that property, which happens to be an entire object. And then from there, it's accessing the property full, or it's creating the property full, excuse me. And it's assigning it the value true. And that's where we get that value. Does that make sense? Thumbs on that? Sideways. Do you have a question about it? What are we doing? We're nesting what? We're putting an object inside of our object. But why, why? Why? Just like, you're going to be dealing with nested data structures a lot, and you're going to be creating them, moving them around, getting data out of them, organizing that data in a way that's meaningful for your application. Is it similar to having an array of objects? Yeah, so you can have an array of objects, you can have an object with an array that contains more objects, and you're going to find that you're going to have these super-nested data structures that you're going to get from your API, and then you're going to have to turn that into DOM, and make it a user interface. But first you need to learn how to deal with a nested data structure before you could even get there. Is there a lot of the more complex, let's say, nested structures, is there any way to, because you'll see a wall of text, that's your JSON. Is there a way to run something that see what that structure is, you know, without the values in it? Typically it's a bunch of these, or is that-- Right, so if you're getting it from an API, hopefully documentation might have some information on that. If you have an in point API where you're receiving the data, you can actually just directly go to that URL, and it will print out the data for you directly, and then there's Chrome extensions that sort of prettify it to make it indented and stuff. I guess sort of prettified would work, I guess, put it into a tree-like structure. Yeah, so you can look at it. OK. Yeah. Great question, though. Cool. So in again, and again, the rules don't change. You're going to also use dot notation right after using bracket notation. Still the same. So what is this going to be? What is this going to look like? box.innerBox? Squiggly line, full, full, and true? Everyone agree? Thumbs on the answer? Feel like it's been a long day, we're starting to fall asleep. We're almost there, almost there! So close, awesome. So what if we assign box.innerBox to a variable? What does this look like? Same thing? Yeah, absolutely, it's going to be the same. Oops. Full, true. So... what about here, what if we triply-nest our object and now we have a box, we have an innerBox, and then we have the babyBox. We have a little box family. So and then we're just going to console.log this line here. What is that going to... look like? What is it going to look like? Empty object? Yeah. It's just going to be an empty object. But it wouldn't have, like, three squigglies on the left, and three squigglies on the right? No, it's just one object. It's just one? OK, got it. Because the way to think about it again is we first accessed box, right, which is this whole object. And then we say box.innerBox, right, which is, here's innerBox, so the value stored there is this one. And then we say babyBox. And so the value stored at babyBox is this object. This is before we add the property, so it would be empty at that point. So it would be an empty object. And then, if we do it afterwards, Whoops. What is it going to look like now? What's up? Yup. Oops, kind of. Oh. Cool. So the interesting thing about, I'm going to delete this last line. So the interesting thing also about objects is that if we say var bb it's going to be babyBox, equals that. So here we're going to assign this empty object to the variable bb. Then when we add a property to our babyBox, and we check bb, Oh, every time I do a question mark, it does that. What What is that going to log there? It's OK if you're wrong. Blank? Empty string. Empty string? Empty paren, really. Good guess, and that would be true. That would be true, it would be an empty object, if it was a string. So if we did this. And it was a string, and it wasn't actually an object, it wouldn't, um. It wouldn't change. However, because of something we call being passed around by a reference, bb now is going to actually be referencing the same object. So if you add, uh-oh, whoa. So if you add properties to this object, it's going to show up later. And this is true for objects that raise n-funcs, this is not true for primitive values, like numbers, Boolean values, strings, null undefined, things like that, just objects that raise n-functions, anything that's an object. And so that's something to keep in mind when things start to change a little weird, and you're not... Things start to change when you're not expecting it to, and it's an object, you might want to check on this.

More Nesting Examples

So here we have an example of our innerBox, so we just have a doubly-nested object, we can handle this, right? It's only two. And with property full equals true, what's going to happen in this next line here? How about Ben, what do you think? For what, what's the question? What's the line going to do? Set the property of height of the object's innerBox to 10. Good guess. Anyone think something different? Undefined because innerBox doesn't have a quality on it? Yes, good, good catch. Seeing if you guys are awake. There's coffee in the back. (laughter) OK, awesome, so we fixed that. Cool, that we'll pass. What about the last line there? That's not going to work. And so, what my mission, or my request to you is make that last line work, without changing the last line itself, and we want innerBox2 is going to be a different object. So what, there's a few things you're going to need to do to make that last line work. You can do it here, above it, so that that won't throw an error. So I'll give you guys about five minutes to do it. (throat clears) And also note, innerBox2 is a new, is a different object, it's not the same as innerBox. Did you have a question? Did you mean to leave the mouse closed? Yes. So you can't, you can't change this line, will not be changed. Do you want innerBox to be another box? Yeah. Another, like a sibling of innerBox? Yes, exactly. So it'll be another object. So you need to create a new object in there. Cool, so, our online friend Sancar, sorry if I pronounce your name wrong, answered this, so I'll just go ahead, type out the answer. So, to do this, we first need to, to create a variable, right, called innerBox2. We know that because innerBox2 doesn't have quotes. And we can like, call it something else, even. So we can call it otherBox. And then we have to, on box, put a property called otherBox, and set it equal to an empty object. So we need to do this first so that we have an empty object, on box, to add the full property to. So now we say box, which is our object, and then we say, oh look, that's a variable, what, what does that value hold? Oh, it's otherBox, so you can imagine then, that this is swapped out, so that would say like, box dot, or box.otherBox, which we know is an empty object. Here. And then we'll just put a nested property on there, I'm sorry we'll just put a property onto that object, set it equal to false. And that's the answer. Whoops. Any questions on how we got from from just the empty object, from the variable to the empty object? Couldn't you just assign innerBox2 to box? So, point to box itself? Well then, then it would say, then it would be kinda like saying this. Which wouldn't make sense, because we don't have, and you really don't, that's called circular pointing. And that's like, something you don't want to do. But, Yeah, you can't really do that. Because we don't have a property called box on box, right? So you could create it I guess, but. So Kate asked, when do you know to set a variable to an empty object? Do you always have to do that, is it a best practice thing? Well if you're creating an object, Can you repeat the question? Sure, the question is, when do you know to set a variable to an empty object, and do you always have to do that? And is it a best practice thing? So whenever you create an object from scratch, that's what you have to do. There is constructory notation with the keyword new, and then capital O Object, but, that's not a best practice, in general, and so it's better just to initialize your empty object using the curly brackets. So what this does it that it just creates an empty object, so that you can add properties to it.

Nesting Exercise

So this is going to be our last exercise. It's one of the longer ones, so. Let's see, we go back home. Yeah, it's just called nested. Nested data. And, oh I guess it's not as long as I thought it was.

Nesting Exercise Solution

So let's just go through these quickly. So here we are creating just an empty array. And we're assigning it to the variable friends. friends here we're pushing, from our animals collection, the first, the very first value, which is an object that has, that has a property username. So we're adding the username property. In this case we can see that it's Mittens, from the previous exercise, as well as second, or the first index really, value, which is DaffyDuck. And notice that for this syntax, for the .push, you can actually pass in more than one value, and it will just add both of them, in order. Cool. And so here's our relationships objects. So we are in our relationships object, we can have different types of relationships. So one of those might be our friends relationship. And our friends again is just our object with Mittens and DaffyDuck. And now our relationship object has a property called friends, with a value of an array that looks like this. And we can inspect that, or actually we can't, because I don't have previous. So then when you console.log relationships, It's going to look like this object, and if you wanted to count the keys, you could use this handy object.keys method. It will return an array of all the properties. And then you can take the .length. And so we see here that our object, our relationship object has one key, friends, and therefore our length is then one. So we have our friends, and then we also have our matches. So our matches is an empty array, and here, here we are putting our matches array, empty array, onto our relationship object. Add a property name called matches. And this is again with our nested, we keep adding adding on our commands. So the first thing, our first command would be, like, here's our, access our relationships object. Second one would be, access that matches property. And then the second, the third one would be to push this value pig onto that. Then we can console.log our relationships and we can see now that we have a few things in our relationship object. Wrap it. So we have our friends, which is an array with two names and then we have our matches, which is an array of one, with pig. And then if you wanted to loop through your animals, or your farm, you would just do your regular loop, and at animals at i.relationships, you're going to add that relationship object, and you can console.log your animals, and inspect the relationship object. Keep in mind, though, that this is going to always be the same relationship object, so all your animals on your farm are actually going to have the same friends and the same relationships. And that's an interesting thing, to inspect and change things around and see how it changes in all of them. And again that's only true for things like array objects and functions, for... for strings, numbers, Booleans, all those primitive values, that's not going to be true. Awesome, so any questions about what I just went over? No? Okay. Awesome, so that kind of wraps it up for today. If you want the solutions, there's a link in the slides. Submit your email, there's an option to opt in for notifications about upcoming classes taught by me or my colleagues at HackReactor. If you'd like, either way you can have the solutions and also there's a link for live classes that we have monthly at HackReactor. We are going to do this class, actually, on Sunday, so if you know anyone in the San Francisco area that's interested, there's a few spots left there. And, yeah, and you can just check the Meetup for that. Awesome, thank you guys so much for being here. I really appreciated your time and attention, and I hope that you got a lot out of this class, and that you can continue growing with your JavaScript.

Scope

Introduction

(music) So, this class is for someone who has for the first time has just started learning JavaScript for a few months, usually two to six months, maybe you completed an online tutorial like Codeacademy, Codeschool, Khan Academy, something like that. You want to continue solidifying your fundamentals, you're looking for more practice or you're trying to get into Hack Reactor. So I initially created this class to help you prepare for the Hack Reactor interview. It's a two-part class and this is the second part. So the first part which I taught yesterday and will be also released on Frontend Masters just reviews the core foundations of JavaScript. We talked about objects, arrays and functions, those are some of the most important aspects of the language, as well as the most confusing for beginners. And then today, we're going to build on that and we're going to explore even more into functions and get started with some functional programming with the underscore library. And the hope for this class is once you have the strong foundation, you can go out and be an even stronger JavaScript engineer. Cool. And so I'm just going to jump right in review because we have a lot to cover today and we only have like seven hours or so. Here we go. So, here's a chart from yesterday and we have the anatomy of a function. So, what is the declaration or the definition of a function from yesterday? Joe. It's the actual commands that get executed as part of the function. Yeah. So the function declaration is just, starts with the keyword function and it follows through to the bracket. So it contains the function, it contains the para, it contains the keyword function, the parameters and the function body. And what about the function name and note that the function name here is being used loosely to mean any way that we can refer to the function later in our code. So that could be the actual function name, it could be a variable where the function is stored. So we have the function name, this is an add function. So we have yellow add here, that's a function name and we've referred to it down here as well. And then we have the parameters. The parameters are placeholders or variables. They're like variables in that they don't have a value until they're assigned a value. And the way that we assign parameters a value is by passing them arguments. And so here are our arguments downs here and we pass the arguments at call time or when we invoke the function. When we invoke the function is when we pass the arguments and at that point is when our parameters gain a value. Before that our parameters are undefined. And then the other important thing is the function body. So the function body is never run until we call the function. So the function body is in between these two parentheses here and that code that return A plus B is never run until down here. So if we were to pretend that we were in the interpreter we'd start here above this add function. We'd say, okay great, we're going to create a variable and store this function and definition in that variable and memory. And then it's going to skip the entire function body so that return A plus B is never, it's never run. And then once we get to the add functions, oh, this variable and. Oh yeah, it's a function that we're going to call it and we're going to pass these arguments here, three, four and five. Then at that point only does it go into the function body and run the containing code. So, we invoked it down here and we skip back up and then it goes into this statement which is return A plus B and also at that point is when we give value to our parameter. So at this point is when A and B now contain the values three and four. So, that was a review. Do you have any questions on sort of the anatomy or the function, the vocab words? So thumbs. So on those of you joining online, thumbs, a thumb up is a B and a thumb down is a P and then a middle thumb is just a dash. And so, we use the thumb to sort of gauge where everyone's at in the room and we're trying to be honest with each other about where we are so that I can go slower or faster based on how the collective. And then it also just gives me feedback on if I explained something thoroughly enough or if I should explain something a different way. So it's really important to really be honest when we do our thumbs. Cool. Lots of thumbs up down there. Cool. So-- so, the slides, the slides link where it says /live, that is the live slideshow. So you can tune in there or you could just go through the slides yourself.

Local and Global Scope

So the first thing we're going to, we're just going to jump right into scope. What is scope? How many people here have heard of scope? I've heard of scope. Nah. Never heard of scope. What is scope then? How about Andy, what is scope? I'd say scope is from my understanding a set of rules where it defines where variables are stored I guess. Yeah. So, it defines where we can access variables, where variables kind of live or exist. Cool. What about Rich. Yeah, I look at it as like an area where a variable or something is relevant outside of that scope. That variable or element may not even exist or you know. Yeah, absolutely. It's like a hierarchy of maybe spaces if you will. Yeah, absolutely. So, in essence scope is where our variables have meaning. So if we try to access a variable outside of its scope, that variable won't exist and will throw an error. And the important thing about scope is scope is created dynamically whenever we call the function. So we think about scope a lot of times as being in the function body itself and you can think of, you can, the... You can sort of think of these two brackets here that define the function body as a representation of scope. However, the scope itself isn't created until we call that function here. So every time we call a function we create a new scope. So if we call this add function three times we then have three scopes created. And that's why we can have different values in the function body. Using the same function have different values for our parameters for example or anything. You have a question? I have a question. Other structures besides functions but have curly brackets like-- Mm-hmm, like blocks. Took while's and for's, do they create their own scope? Not in JavaScript. Only functions. Only functions, mm-hmm. So, we have this thing called local scope and this is how we create a scope in JavaScript is by wrapping in a function. And this is the only way that we can create a scope is by saying var name inside of a function body. So if for example we console log local out here this would be outside of that scope and that would give us an error because there's no local variable there. And this is also, oh, do you have a question? Question is if you declare a named variable as the function versus a function than the name is it the same thing? Oh, what's the difference? Yeah. It's basically the same thing. So the question is what's the difference between naming a function like this or like this? The difference is that, the difference is that this function name here is going to be hoisted to the top of the scope and that you can refer to this function, the whole function body everything, it will be hoisted to the top. Whereas if you store it in a variable it will follow variable hoisting rules. We're not really going to get into hoisting today, it's kind of like a... I don't know, it's something that happens in JavaScript that's sort of like extra points if you know it but if you just follow the general rules, you shouldn't have to worry about it. But in general I would recommend to store it in a function, I'm sorry, store it in a variable rather than just creating a named function. If you take the V-A-R off of that second line then will the console log work? That's my next slide. So good, you're a step ahead. Sorry. No worries. So this is a private variable and local is in the local scope and then we have something also called global scope. So global variables can be accessed anywhere in your application. This could be dangerous if you have naming collisions. You might not know which one wins then you can have unexpected results. So, in general you want to avoid putting things in the global scope. So one way to create something in the global scope is to simply outside of a function say var x equals global. Or even if it's inside of a function what you could do is you could leave the var off. This would still work. It just automatically makes that variable part of the global scope. So make sure that you do have those vars. The indentation there is not relevant in JavaScript? No. Always space doesn't matter. Okay. And another thing that's kind of a hack is that you could put it on the window object so all global variables are located on the window object. So you could say window dot y, or at least in the browser. So on browser land, everything is, every global variable is attached to the window object. So this wouldn't be true in different other environments like node or if you're programming like a TESL or something. The different, the environment depends on this but in the browser window is where all of our global, globals live. Mm-hmm? So that z equals global here tool, that's not, that's global because you didn't use the var? Yes. Any other questions? And I could access x or z with window dot x or z later on if I wanted to? Yes, yep.

Parent vs. Child Scope

So we're talking about how to make privacy with or make local variables in our locals-- in our functions but what if we have a function nested inside of another function, how does that affect our scope? What do we need to know about that to effectively handle our functions? So, we have here just a review is we have this blender function in here. See, this is a name function. This is the one that's being stored in a var. So we have a blender function, it takes fruit and it assigns that fruit to b, and then it blends smoothie, bs means blend smoothie. And it's going to alert the fruit, whatever you pass into it plus yogurt makes a fruit swirl. And then we call it here and then we call it down here. And the question is, what if we, like where can we access these variables in this nested scope? So the parent that has no access to the child scope, so if we said var, I don't know, x equals astf. Sorry. My creative juices are a little, are a little slow this morning. We couldn't say down here. We couldn't console dot log x. That's because while child scopes are nested, the inner nested function has access to the parent scope as well as the global scope where the parent don't have access to the child scope. And at Hack Reactor we use this metaphor for functions that I mentioned yesterday where a function is like a blender, a blender is an object. We can move it around, I can hand it to different people and we can say that in a variable if we wanted to. But to run the function or to turn on the blender, you have to put in the fruit and then press the button, right? And so, call time is pressing the button for example and when... And so, the extension of that metaphor as it applies to scope, it's a little gruesome but you never want to put your hand inside the blender. So if you kind of forget which way, who can access what, if it's the parent or the child, just remember you can't put your hand inside of a blender or you don't want to put your hand inside of a blender. And that will help you remember that the parent can't reach into the child scope but the child can actually reach into the parent scope. So that's why here in this alert we can access the variables in the parent scope here y and b but we can't console dot log x down here at the bottom. Mm-hmm? I guess I'm confused with the analogy. How can the, why would you want a child reaching into a blender? (laughs) You don't want children reaching into blenders. It's not what I'm trying to promote. You don't, so this... So we have a nested function here, you see that we have this blend smoothie function and then we have the blender function, right? So we have this function inside of the function and the question is which scope? So we have two different scopes that we're talking about. We have the parent scope and then we have the child scope, and we're trying to figure out which scopes have access to which variables? So the child one, the inner function has access to the parent scope and the global scope and all the way up the chain. So if it was nested many times, any parent, it has access to those variables in that scope. However, the parent cannot access the child scope so this outer scope here can't access this inner scope. And that's why this console dot log x would throw an error. That makes sense? Cool. Somebody, I was just asking I think of why you were calling the bs function rather than returning? Rather than returning it? Yeah. Just because we're just calling it, I mean there's no reason. We could return it but that adds an extra step of having to call it later. We're not returning anything from this function at all, we're just alerting, so we could play with this function if we wanted to. So, I don't know if you can read that but it says blueberry and yogurt makes a blueberry swirl. And so, we just created the scope here. So when we called this blender function we created this scope that you can kind of represent by these brackets. We created a scope there. And then here, I'm sorry, and then here when we called this blender, blend smoothie function, bs is like a terrible naming. I didn't think about that ahead of time. When we call the blend smoothing function, we create a second scope here. So this inner scope has access to the parent scope and that's why b and y, they know what values they hold because the child scope acts as the parent. If you move the bs invocation down two lines will that work? No. And the reason, so the question was if we move this, this bs call time down here, would it work? And the answer is no because the function bs is actually local to the blender function and so just like we couldn't console dot log local up here because it's private, we can't console log. We can't access the bs here. So and the other thing is there's no arguments being passed to bs but it's still rendering the blueberry and the yogurt because they're-- Because it has access to the parent scope. Has access to the parent scope. Mm-hmm. A follow up on the moving the invocation. If you move it down two lines could you say blender dot bs? No. No, you couldn't. But if you named it that way wouldn't it hoist it up? It would hoist it to the top of the scope. So it would be, you can-- Okay. It would be like equivalent to just like... So how the hoisting would work is it would be an equivalent of just like putting it up at the very top. So we could technically, oops. We could technically call it up here. This is like not something I want to cover too in depth because it's a little bit confusing. But the way that this function works it's hoisted so that you could actually call the function up here. Yup? Question about the variable x. If it was declared with the var keyword would it be in the global scope or just the scope above it? Good question. So if we do it like that and we take off the var then that would be in the global scope. So it would be the equivalent of being in the scope up here.

Precedence

And so this is also how we create privacy, private variables is by storing them in a function and there's no other way in JavaScript. We don't have block scope like Ruby does for example. We have a lot of people who take this class who come from a Ruby background and that's a little bit confusing at first. But there's only, only, only, at least at this point in time there is in future generations of JavaScript. There's going to be a let keyword that will allow you to have block scope but for now we don't. Cool. And so I spoke a little bit about creating scopes, about how the scope is not created or variable scope is not created until we invoke the function. Cool. And then so we have these nested scopes. How do we know which one wins when we have two variables with the same name? And so this is called precedence. I like to play this game where we go around the room and we pretend that we're the JavaScript interpreter and we say what line we start in. So for example what I would say if I was going to do this previous example I would be like okay, so I'm JavaScript interpreter, I just created a variable g. I stored the string global. Then I would see the blender. I would skip over the blender. I would store this in memory. Skip over the blender then I would go down this invocation where I would call it with blueberry. That's when I come into the function body and the function body fruit is now blueberry. So b is blueberry. Then we're saving y as yogurt, a string and then we see this function. We skip over the function body until we get to this last line here in which we call a function. And at that point is when we go into the function body and we alert, blueberries and yogurt make a blueberry swirl. And then at that point this function is out and then it comes out here, and it goes into the next line if there was a next line underneath. So that's kind of how just sort of like going through how the code would run. And like a very crude understanding of like how it's run. So, who wants to go first and then we'll just go down the line? The first line is the easiest one so it's better if you go first. I'll take it. Okay. All right, Grace. Go Grace. So g is global, I mean I don't know what else to say about it. Yeah. So you're just storing a string global in g. Yeah and then Joe. Var one is local. Is that null? So the next line that it runs is going to be this line, right? So what am I supposed to say? Just how it would run. How the code is actually going to be run. So you could just say we, you could just say that that there's a function. Function call go. Function called go, yeah. So you store that in memory and then you invoke go. So we skip the function body, right? Yeah. Skip the entire, so this part we skip. That's really, really important I'm going to say like eight times but that's super important. And then we. And then you call go. Call go and then Kim, what happens next? L equals local. Yeah, we go into the function body. L equals local and then William? Is your name William? Yeah. Okay. (clears throat) Then you would be on the next line for the g variable, you'd set that. Mm-hmm, in here. And then Andy. Then you're going to alert g plus inside go. Mm-hmm and what's g? G will be the local variable in the go function. Mm-hmm, so it would be in here, right? Yeah. Great. And then Kevin? You're going to alert g outside of go. Yeah and then we'll, so once this function's done running it's going to return back to here. it's going to exit here and run the next line which is going to alert. What is it going to say? What's it going to say? Outside go. Well what is with the g part? In here or global outside go. Yeah. So this is just an example of how the code is running to help you understand that whatever variable in the most local scope wins precedence. And that's the variable that's, that's the value that's going to be printed out. So since g here is the most local to that scope, g is going to overwrite any memory of any other variables if there was, if there was any, and it would alert here. And then the most local to this global, right, is going to be the global g. Any questions about that? Joe? Is the whole... Does go come off the stack then after the execution is done? Yes. Cool. And so, back to block scope. Let's play the game again. Where did we leave off? We left off at Ben. So, what's the first line? What happens the first line? Create the variable in block and set it to false. Mm-hmm. And then Jake? I'm sorry. He said which one, the top? Yeah, he said the top one. Okay so, we skip that and we go to the, the bottom correct if in block. Well, we have this one still. Oh, okay. So this is a for loop, this isn't a function. It's a little bit misleading because it does have a block. So a block is defined as any two, any area surrounded by two parentheses. I'm sorry, two curly brackets that's not an object. So whenever you see in JavaScript the syntax where you have two curly brackets and it's not an object, that's considered a block. In a lot of languages, a block would define a scope but in JavaScript only a function block defines the area of like private scope. So this is the line here. Do you know what this line does? No? No worries. So, it's going to loop, right? It's a for loop and what does for loop, it says start at zero. Keep adding one to i until i or while i is less than five. So it's going to run whatever is in this body that many times. So it's going to set in block to true five times just to make sure that's true. And then what's the next thing that run. I'm sorry, I never... Oh, Tanner's next. Okay. Okay. Next we go on to if in block and the console log which is it's going to pull up the false even though we set up five times true because it's out of scope. So what is if in block going to do? Sorry, I skipped over that pretty quick. That's going to invoke the console dot block and it will ask if there is block scope and then it will add false because the other variable in block is out of scope for the first one. Absolutely. So this if is going to evaluate whether in block is true or false. If it's true it's going to enter into the if block here and then it's going to console dot log. Is there a block scope and this is not, so this is going to return the opposite since in block is true and this is not, this is going to evaluate to false. So the question is, is there block scope in JavaScript? Well, I don't know, let's ask. False. Mm-hmm? Would it ever get into that block though since it's false? The block? It's true though. We set it true here. Oh right. And since we don't have any, so that's all the global scope. So it's all, in block is just going to be true. So it's not private. So if this was a function, if this for loop was a function, then when we changed in block to true it wouldn't affect this in block up here. But since we don't have block scope and this is all operating on a global scope, that's why we can, well, that's why this is true here. So, var in block equals false, that's a global variable. Yeah. They're all, everything on this-- Because there's no child, there's no-- There's no function. So, everything on this page is global. It's global. Got it. Mm-hmm.

Scope Exercise

So I have the exercise. We go to the very first slide. We can have, we have the bit.ly which is func exercises. Mm-hmm? There's a question I'm not sure. Do you need the var in the news that said maybe the for in block? Oh, the question is do we need var? Yeah. No, you don't need var. I didn't want to give it away though. So, question was do we need var here? Oh, I can't delete that. But do we need var, this var? No, you don't. That a red hearing? What's that? You don't need it but it's-- If you put it there it's not going to matter, yeah. So I asked if you could touch on the let keyword. Basically-- We cover let in very, scope and let in the advanced JavaScript course on Frontend Masters. Yeah. So, if you don't. Yeah, basically with let it's just like instead of using var you could say, oh, I can't edit this. Instead of var here you would say let. And so you say let i equals zero and then, then dot block, i would be in that block. But I've never used it before. I don't have any experience with it but that's the idea. And if you take the advanced JavaScript course it will go more in depth on that, more in depth on scope, like you can... There's lots of aspects to scope and you can really dive deep into the mechanics of how the JavaScript engine works. And like how it parses it and goes through and more in depth, and it's really, really interesting and it will really inform like your understanding of how JavaScript's really working in the background. The way I do it is just kind of like willy nilly, kind of just the basic idea of how it's run but there's a lot more to it actually. Cool. So, here is a link for the function exercises and we're going to start with scope. This is a little different than the exercises that we did yesterday. The scope exercises are test driven. I have a doc here, the read me that will explain exactly how to do it but the basic idea is, let me see if I have it open. No. The basic idea is if we go here is that you're going to... You can copy this. You can clone down the repo if you're familiar with that or you can just download the ZIP. Once you download the ZIP you can go into the scope folder. You open your spec runner and your browser and that's going to show you a bunch of failing test. Yeah, I'll just (mumbles). I'll just show you how to do it, let's see. Okay. And then let's see. So I just download it. I only download, I just have in your unzip it. Scope exercises. Where'd it go? Oh, here it is. So inside there we have our spec runner which we open in the browser. And you can see that we have one pass and 11 failures. And then see here. Then we want to open up this function dot js and you're going to open it with your Text Editor. In here we have some where our failing tests are. Let's see if I can. So we have this describe block which is going to encapsulate all of our tests and you see that we're running it immediately down there at the bottom. And then inside of the callback function here, we have before each. So before every it block, this before each is going to be run and we have this var actual variable that's just initialized to undefined. And so before each is run actual is going to be set to null and then inside of this it block, we're going to explore the rule that this string describes. So this one says a function has access to its own local scope variables, and in this callback function we're going to be testing that. So here we have this function and then we have a variable named inner. We set actual to name. We call that function and then we have this expect statement. So as we expect actual to equal inner to be true. So we want this to, this conditional here to be true. So, you could just actually type in true but that would cheating. But what you want to do is go through and change these three question marks to what it would be, either it would be null or undefined or some string. And so you go through and then to check your answers you can... Yeah, let's just make this one false. You save it and then you go to your spec runner and you refresh your spec runner and it will show. So we can see that this one didn't pass. And then if we wanted to actually pass we'll just change it to what it is. We save it and then we refresh the page and we see up here. Can you guys see that? See up here that we have one pass. And this will show you the rest of the ones that you need to change and it has a little description of the error here.

Scope Exercise Solution

So I'm just going to quickly go through the answers for this. So it starts off with 11 failures and one passing. And let's see. So I'm going to walk through it with you. So, here's one insertion, it says a function's local scope variables are not available anywhere outside of that function. So here we have this function called first function and we call first function here. So then we enter into that function body. We set local first fn to the string inner and then we exit the function. All this is doing here is it's stopping it from, whenever you have an error, it will stopping running your code and will just break everything. This will stop that from happening so it will run the subsequent lines of code. And this one says expect actual to equal what? So we're setting actual here. Local the first function. What string? The one that-- Inner. To inner. The one that's right there. Like this? Oh, a string. So we can check it, so I just saved. So I just saved that and we can check it by coming into our spec runner and refreshing it. And we see that that didn't pass. So, there's some assumptions here that we-- Why did you skip the test above? Oh, did I skip? Oh whoops. Sorry. I'm in the middle of the screen. So sorry. Let me start from the second one. So this one's the first one, here's the second one. So the first one is inputs to a function are treated as local scope variables. Is that a true statement? Yeah, so you want to... It's a true statement and then the code you're trying to prove it right. So it's testing for this rule. So we have this function and it takes a parameter called name. And then here we're calling function to equal with inner. And so, the parameter is now holding the value inner. And so actual equals inner. So we type in inner and then we save it. The saving part's really important. And then we go to our test and then we refresh this page and we see that it passed. Cool. So and then we're... And so the point here is that parameters to a function or I'm sorry, arguments. Whatever your input is is going to be local to that scope. So a function has access to the variables contained within the same scope that the function was created in. So here we created this variable outer. We have this function here that we call and we say actual equals name. What do you expect that to equal? Outer. Outer. The kid can reach into the parent but the parent can't go into the blender. Yeah, exactly. No parents in the blenders. We don't want that. And then we can refresh it, awesome. So that passed the test as well. That test passed as well. All right. So, now we're back to the other one. A function's local scope variables are not available anywhere outside that function. So here's a function, first function. We set that variable, that local variable to inner. We call it. And then here we are, this is kind of a hint that it's expecting it to throw an error when we call this function. What is this going to be? Null. Null. Why is it null and not undefined? Because it's not declared anywhere in the global scope or in the scope outside of first function. It's a good guess. Is it because of the expect thing? It's so that the code doesn't break. Actually it has something to do with this before each. So before each of these it functions, it's going to set actual to null. Right but because that expect doesn't actually run the code to set it to undefined, it just keeps it at null? So yeah, every time it will set it to null. It will reset it to null every single time. So if we didn't have that function there it would, actual would, wouldn't, would keep holding the previous value of actual from the other exercises if you're not careful. No, I understand that but to make this exercise correct, the one we're on now the actual is expected to be null. But the only reason is that actual really hasn't changed because the before function changed it to null. Yeah, exactly. We started as null. Nothing's changed it. Totally. And the reason nothing's change it I think, I guess my question is, is because you have that expect statement that kind is like if you catch it, you know. If it's going to be undefined or an error then just leave everything the way it was. Right, it's because, yeah exactly. Because this local, the first function, since it doesn't exist in the scope it will throw an error Right. And it would stop running the code. So (mumbles). So this actual doesn't have access to this variable. Right. Okay. Yeah. She didn't have that before, what was it called? Before each test at the top. It would be outer because outer was the result of the previous test case. Mm-hmm. Oh, okay. Yeah. Got it. So the next one is a function's local scope variables are not available anywhere outside that function regardless of the context it's called in. So we have a first function here. And we have this variable first and there's a little hint here, it says, although false, it might seem reasonable to think that the second function mentions the local. Second function should have access to the local first function variable since it's being called here from within the scope that that variable is declared. This is a tricky one. So, we have our second function here. Oh sorry, we call second function here but since this hasn't run yet we actually even haven't gotten there. So we declared second function, actual equals local the first function and then we expect this to throw. So this is where we're calling our second function in here. Oh I'm sorry. I actually think it's going to call first function first here and where that goes it goes into this inner block. Declares this local to first function then it calls second function and then second function is setting actual to local first function. And this is tricky because this is something called lexical scope if you want to read into it later. Is that this doesn't have access to the local first function. So what do we expect this to be? Null. Let's double check it. So blenders can't access sibling blenders? I mean their function wanted second from first like siblings if you will. Yeah, you could call them siblings I guess but the point here is that yeah. So, yeah siblings can't access other, inside other functions either, yeah. Even though it might seem like you could because you're calling it from within that function. Cool. So if an inner and outer variable share the same name and the name is referenced in the inner scope, the inner scope variable mask the variable from the outer scope with the same name. This is from the slide where I called precedence. This renders the outer scope variables inaccessible from anywhere within the inner function block. So what's this one? Oh, so we have a function or sorry, we have this variable outer and then we have this function that has the same name as we set to inner and we set actual the same name. What are we going to expect actual to be? Anyone? I hear whispers but I can't understand them. What about Kim, did you get this far? No because I didn't know what role meant. I don't dig into testing so I don't know any of that stuff. Yeah, yeah. No worries. Anyone else get this far? I would say outer. Outer? What did you say? Inner. You think it's the inner. You're right, it's going to be inner. Why is it going to be inner and not outer? Because it only gets as far as the. Because if we think about the way it's called. So we call this function. Yeah. So first we set outer up here and then we say awesome, there's this fn function, we skip it. We call it and then at this point we enter into this, to the function body. Now we set same name to inner and at that point we're setting actual the same name and at that point it's inner. And that's why when we, when it's evaluated down here it's inner. I was looking at the bottom at the next exercise. Oh okay. Looking ahead. (snickers) Awesome. So, the next one is if an inner and outer variable share the same name and the name is referenced in the outer scope, the outer value binding will be used. So, here we're declaring same name, great function. We skip over it, we call the function. We enter into the function body. We assign same name to inner. We exit here on line 85 and then we set actual the same name. And since actual is in a scope that's outside of this function, it only has access to the same name within the same scope. And so that means actual is going to be. Outer. Hmm? Outer. Outer, yes. Then we save it and then we can check it to make sure we're right. Awesome. We're starting to get more greens. So a new variable scope is created for every call to a function as exemplified with a counter. So we have our fn function. We say cool, we store it. We skip it and then we call it. And then we have, we call it, we enter into the body. We say var inner counter equals inner counter or 10. And so, if inner counter... So if this is undefined, it will be false and it will skip to the right side and it will be 10. Otherwise, inner counter will be inner counter plus one. Do we have an inner counter? Okay. Oh, I see. So, if there's no inner counter already then it's going to be 10. And at this point it's going to be 11 and then we set actual to inner counter. So we call it once and what is this going to be? 11. 11. 11. And then we call it again. 12. 11. 11. So it's going to be 11 because every time we call the function it creates a new scope. And so inner counter is never going to change because it's always going to start off as undefined. Great. So a new variable scope is created for each call to a function as exemplified would uninitialize string variables. So here we have our function again and we skip it right to the end and we call it. We go inside and we say, we initialized a local variable at undefined. And it says if local variable is undefined, enter into this block. So actual is now alpha and then we're going to skip the else. And then we're going to set this local variable to initialized. And now we're going to call the function. The first time you call it what is it going to print out? Alpha. Alpha. Alpha. Exactly. And then if we call it again? Alpha. So if we call it again. That's initialized. Well, let's think about how it runs. So we call the function, we go back up here. It's initialized again as undefined. So it says if local variable is undefined enter in this block. And it says actual alpha, it skips it and then it says local variables initialized. So, mm-hmm? Sorry I got tripped up on this one. I thought local variable's null instead of undefined. Could you maybe touch on why it's not null? Sure. Sure. So, the reason that local variable is undefined is because whenever you don't explicitly define a variable, it's undefined, you have to actually set something to null. So, JavaScript will make it undefined automatically but... So in general if you want to set something to empty it's best practice to set it to null and not undefined so that you can kind of differentiate between that. And you'll notice that we set actual to null before each time. That's us manually setting it. But here we're just initializing a variable and it's just going to be undefined. The same with if you pass, if you have parameters that you don't have an argument pass and you referenced those parameters, they're going to be undefined. Yeah. So what did we decide this one was? Alpha? Alpha. Cool. So, oh let's check it. Awesome. So an inner function can access both its local scope variables and variables in its containing scope. Provided the variables have different names. So, here we have our outer name then we have fn again, we skip it, we call fn. We enter into that body. We set inner name to inner and then with actual is inner name plus outer name. So what is actual? Inner outer? Mm-hmm. It's inner outer. And the reason that actual can be set like this is because it's a global variable and we see that it has no var. So that's something important to note as well. So we check it. Great, only two more. Let's see. Between calls to an inner function, that inner function retains access to a variable in outer scope. Modifying those variables has a lasting effect between calls to the inner function. So here is our outer counter and we have a function. We skip it, we call it. We enter into the body and outer counter equals outer counter plus one. So outer counter is 10. So that means outer counter is now 11 and we're setting actual to outer counter. So what is actual? 11? Yeah. And then we call it again. Call function. We go inside the function body. Outer counter, what's outer counter again? 11. 11 plus one. There's 12. So now outer counter. I'm sorry, now actual is set to outer counter which is 12. So what should we expect actual to be at the second call? 12. And the difference here is that our counter is outside of our function. And since our child function here, our nested function has access to that outer counter, it still it can change it but it still has reference to the value. Cool, let's save it. Great. And then our last one. The rule about retaining access to variables from the outer scope still applies even after the outer function call that created the outer scope has returned. So we have our outer function and remember we skipped over it. We call. Oh, where does this end? Oh, I see. So we skip over it down here and then we run the outer function. I call it and we go inside of the function body, we set counter in outer scope to 10. And then we have our incrementing function which we skipped. We call the incrementing function and we say counter in outer scope plus one and we set that. And then we return here to our expect statement. We expect actual to equal what? 11. 11. And then we call our inner incrementing function again and then what happens? 12. 12. So this is like the pre, up to now it's like the previous exercise. Mm-hmm. It's similar but the difference here is that it works explicitly inside of a function. And now what is, what is this one? To be a function. Let's see. Oh, here we go. So, I skipped this line. So then we have window dot retained inner function equals inner incrementing function. So remember we put things on the window object that makes it global. And then we call it here again, retained inner function. And so once we call it we go back in here and then we do plus one again. And since we still retain value in its parent scope. What is it? It's 13. Cool. So any questions about that exercise? It's pretty in depth. If you read through all these rules in here, these are basically all the rules for scope. So if you go through those a few times you'll have be mastered all the different scenarios for scope. Any questions? I like just one question so. And if you click on them? Someone gives me one question that would be great. So if you click on them you get the actual? Oh yeah, if you click on them you can see the exercise. So I have a question. Mm-hmm? I have one error remaining, I don't know which one it is. Mm-hmm. How do you get, how do we find which one it is? Check the red X next to it. Yeah. I can look at yours if you want, yeah. Great. Click the red X. What red X? The style sheet. The style sheet. Yeah, the style sheet, there's an error of the style sheet.

Closure

Closure Introduction

(music) Anyone think about any questions, some scope questions during the break? Yeah. So when you have nested scopes, is that same thing as, I looked it up a little bit, is that the same thing as lexical scope? When you have nested scopes? So, lexical scope basically means that if you look at it, the scope is defined by how it looks, like so if the nested scope is inside of it, it's yeah, so I guess you could think of it that way, like nested scope is an example of lexical scope. Yeah. So if you have the function sibling and even though you're calling it from inside, that scope doesn't work because of something called lexical scope. And the way I think about it is that if you see the function actually inside, then it has access. It's not like an academic definition, that's just like my working understanding of how that is. Cool, and Joe, you said you had a question? You answered it, awesome. Cool, anyone else have any questions? All right. So we're gonna talk about closure. So closure is the next step with scope. A closure happens when you return a function from inside of the function, and that inner function retains access to the scope. We saw that a little bit in our exercises here at the very bottom. So even though we made this inner function global, it still had access, and it remembered, quote unquote, that we had incremented our counter inner scope to 12. And so-- Why? Why? We're gonna get there. We're gonna get there. We'll walk through it. And so that is an example of closure. Isn't it a violation of scope, though? It's a violation of scope. No, I think it's taking advantage of the power of scope to do some interesting things. Didn't we say that the variable inside the function is not available to something that's outside the function? But if you return a function from inside the function, that function was originally inside the function. But we'll get there, we'll walk through it. We'll walk through it step-by-step, and talk about what I mean when I say closure. So our first thing I want us to look at is a function called closureAlert. And it says, so function that has a variable that's initialized that says, "Help, I'm a variable "stuck in a closure!" And then we have this inner function called alerter, and all it does is alert X, and then we call alerter from inside. So let's play with this in our console. Can everyone see that? Even in the back? Cool. Thanks. So now, so we have closureAlert. So if I just type out closureAlert, what does that, what's gonna do, what's gonna happen? How about Ben? What's gonna happen if I just type out closureAlert like that? It's gonna return the definition of the function. Absolutely. So it's just gonna return the definition of the function. And then if we wanna call the function, we put the two parentheses. All right? So we call it. Once we call it, it's gonna go into this function body, right? It's gonna declare this variable, it's gonna say, great there's a function, then it's gonna call alerter, and then that point it goes inside alerter's function body, and alerts X. And then we have this alert. Can someone read that? "Help, I'm a variable stuck in a closure." Thank you. Awesome. So that seems comfortable. We've done that before, right? In previous exercises. We're just calling a function from inside of a function. And it's not a surprise to us that this X refers to the X in the parent function, because we know that their child function retains access to that parent. Cool? Any questions on that? Thumbs? Cool. Did you have a question, with the side thumb? No, I'm good. I didn't see where you called the closureAlert, so that never did call it at all then? Well, I just called it in the console, but for example, you could just do like this. Yeah.

Closure Introduction, Continued

So here's an add on to that example. The difference here is that we have a setTimeout. What does setTimeout do? How long that alert will be done. What would it do? How long that alert will --go. Yes, so it will wait 1,000 milliseconds, which is one second, and then it will call this function. So it doesn't call it right away, it creates a delay. And so if we call this function here closureAlert, let's walk through it. So from the top, from the top let's see. Remind me of your name again, is it John? John, yeah. Okay. So let's walk through how this code would run in the browser. Just one, everyone gets one line. Just one line? Mhm. It's gonna see the closureAlert, define function, and-- Mhm, and then Rich? (mumbles) Mhm. Then it jumps past all that, right? To settimeout? So this is the entire function body. Oh that's the entire function, I'm sorry, okay, so it goes after that and then it calls closureAlert and it milks that. Mhm. And then Grace? So we're in the body of the function and it assigns the string to X. Yep. And then Joe? There's another function there but it gets skipped, right? Mhm, yeah we skip it. Skip this function. And then you settimeout, wait 1000 milliseconds-- Does it wait now? Yeah. Does it pause? It pauses one second. This is actually asynchronous, so it's not gonna pause. It's not gonna pause it. But that's a common misconception. It's like a sketch. Yeah, so it says okay, call alerter in one second. But I'm gonna do whatever else-- But I'm gonna keep running, it's gonna keep going. And then, Kim? Console (mumbles) Mhm. And then William? I'm sorry, I was looking at the comments, where are we at? Um, right here, right at this console.log. All the fun stuff's over. I know. So then where does it go? It'll jump out of running that function, go back to where it's called the closureAlert and... And then? And in one second, what happens? Then alerter is called finally. Yep, alerter is called, so then what do we do? Kevin? Alert X. Mhm. I'll get to the question in just a second. And then, so, alert X. And what's X, Ben? Just the string? Mhm. It's defined earlier in the closure. Yep, what does it say? You can read it in a funny voice, if you want. "Help, I'm a variable stuck--" That's your funny voice? (laughs) In a closure? All right, that works. Just a note here, everyone can please raise their voices a little bit so they can hear. Oh okay, yeah. Cool. So we alert it, and then where does it go, Jake? Where's it gonna complete? It's all complete so go down back. Yeah, and then it just goes down here. Yeah. Awesome. Tanner? So, do you ever have to worry about setting, like, so in that case you set it as second, you're gonna have to worry about the code catching up to the timer that you set, so that stuff might get out of order? Like in this example, let's say you did this on a 386 or something. On a what? On a really old computer. Uh-huh. Would it ever get to the point where it accidentally did get that far? Do you see what I'm saying? Like it would be too slow? Right. I don't, I don't know. I've never had a, I've never done anything where the millisecond is really important, like something down to the millisecond, but I can't imagine that it's so accurate, because of the way that the Q and the stack and the event loop work. So the event loop doesn't run all the time. So the way I, I imagine that it wouldn't be 100% accurate. So. If you had a loop here that was looping over code that was executing without ever returning back to the event processing loop. If that took five seconds to run, settimeout would then run as soon as that event loop is rescheduled. Yeah, okay. And the second after it finished running that five-second process. Yeah, exactly. And then you had a question. The question was, what is the settimeout? Is that a function? Yeah. Where does that come from? Settimeout is just a native function that comes, I believe, with the browser. And we know it's a function because we see these two parentheses next to it, and that's calling a function. We call it. What are some other times that we see two parentheses near each other? So we talked a little bit about, we saw two squiggly brackets near each other. When else do we see this syntax? Expression statement. Specifically? Like an if. Yeah, like an if. So we have an if-statement. We use it, so if, parentheses, and then we have the curly brackets. I didn't hear the question. Oh, okay. Loops. Loops, so we sue them for loops as well. Where else? We use them in functions, right after the function keyword here, or the function name. And that's all I can think of. We use it on a mathematical expression. Oh yes. To assert precedence. Yeah, totally. So we can use it to change the order of our math, the way it's, I guess assert precedence is a good way of saying it. Yeah, so those are the only times I can think of on the top of my head. So we have if-statements, we have loops, there's also switch, and then in the functions. Those are like the, off the top of my head, the only times when we see that. And so just keep that in mind. If it doesn't look like a loop, you know, we know what loops look like. If it doesn't look like an if-statement and all these things then it's probably calling a function. What about when you want to create an anonymous function? Mm, mhm. Is that just like a hack? Or is that valid Javascript? That's valid Javascript. You can use an anonymous function. Mhm? I think I'm lost. Sure. If I were the interpreter. Mhm. The first thing I would do is skip all the way to next to the bottom line. Yes. Then run closureAlert. Mhm. Then I would go up to the top, run the function out of ZX, skip the next function. That way you can settimeout out, wait a second. Then I would run the order function, I would go back up to ZX-- I'm gonna stop you there. Let me just stop you there, because settimeout is not blocking. Say it again? The settimeout is not blocking, so it will actually run, it'll run the console.log next. Oh, I see. An then it will actually just exit this function, and then it will wait one second down here, and once the settimeout works, that's when we get into the alerter function. So it runs in parallel (mumbles) So, what does this have to do with the closure? I can't put the two together. We're getting there. Oh, okay. We're getting there, yeah. I'm just starting off with some things that don't seem too unusual, that we've done before and we're comfortable with them. And then we're gonna get in closure I like to, you know, start off slow. And then we'll get there. And then I just out of nowhere trick you, just like hah. Slams to the next assignment. Yeah. (laughs)

Creating a Closure

So here is the same thing, except, hold on, I like it when it says "I'm stuck in a closure." Let's put that there. Oh, I see, that's why I changed it. Never mind, I can't do it. Darn. So here we have our closureAlert. The one thing that we did change here is we changed X to zero, because we're gonna be changing it when we alert, and the other thing that we changed is instead of calling it inside of here, we're just returning it. This is really similar from our scope exercises as well, right? So we return alerter, which means that when we call closureAlert, we enter into the function body, right? First thing it does is it initializes X at zero. It sees that we have, oops, it sees that we have this alerter function, skips over it, and then returns it. And so our funcStorer is storing alerter. And let's just put that in the console so we can, we can look at it. So if we just type out funcStorer here, we see that it's just that function, right? And the same with funcStorer2, again. And what it's doing is incrementing X and then returning X. That's what plus plus X means. So you add one to X, then you return it. Whereas X plus plus is you return and then you add. So it doesn't show up as well. So if we wanted to then call alerter, how would we go about doing that? funcStorer and then vacation? Yeah, so we just have funcStorer, and then we call it just like that. What's it gonna alert? One. Yep. And then if we call it again? Two. Why is it two? It remembers the scope that it was in. Mhm. And so this is our closure. Which is a fancy word. Just forget the word for now, but for those of you who are wondering, that's the closure, and I'll talk about it a little bit more. So in funcStorer, we call it, it's holding reference to this alerter function, so you can kind of imagine that when we call it, it goes into this function body, it increments X, then returns it as an alert, and so X is then one. If we call it again, and I see you, I'll call you in a second. Call it again. So this is one. It's gonna go back inside alerter, and it's gonna increment again, and alert it, and it's gonna alert two. Is that because we're really kind of storing it, we're not calling closureAlert, we're storing it? Sort of reference it? Yeah, yeah, so we're storing it in a variable so that we can access it later an call it later, and that function, because of lexical scope, has access to its parent scope still. So in a way, it remembers the environment. Actually, you are calling the closureAlert, but the closureAlert is returning a function that's got a defined determiner? Mhm. But you haven't executed that internal function yet. So the difference would be, if we, on the console, just invoke closureAlert twice you just get one every time you do closureAlert. 'Cause you're not storing it. You would get a new copy of the alerter function as the return value. Mhm. So, if we did funcStorer2, that's still gonna alert one, because every time we call a function, it creates a brand-new scope. And did you have a question, Mark? They just wanted you to cover a pre-increment versus post-increment. Mhm. And then the other question was, is returnAlerter the same as calling and opening alerter? Okay. So as far as the incrementer. Here, plus plus X what that does is it adds one to X and then returns it. Versus X plus plus, it returns X first and then increments it, but that's not important. By returns, you mean make available to the alert function. Yes, yes. So if we changed it, and it would first, if we did this it would stay zero, and it just isn't very good for my example. So. And then, the second question is, is returning alerter the same as invoking it and returning it? And the question is absolutely not. Because if we're invoking it here, then we can't call it down here. What we're doing when we don't invoke it is that we're just returning the function definition itself. So if instead of returning it, if we, right here we can console log it here, console.log, alerter, and this will show us what we're gonna return. We'll put this in the console and kind of look at it. So again, we're just console logging Alerter, and then it's gonna be returned. And so we know that whatever this function returns is then gonna be stored in this variable. Oh yeah, and then there's our console log. There's our console log there, and then we can also look at funcStorer, and we see that it's the same thing. And since it's a function now, we can call it. If it wasn't a function, we couldn't call it. So for example, if we just returned alerter and called it already, it's gonna return whatever this function is returning. So in this case, it's not returning anything, but if we wanted to put a return there just for the example, we can return true, so the change here is inside the alerter function and returning true, and then instead of just returning the function definition, I'm returning and invoking the function. So it's gonna invoke the function and return whatever that function invokes. Cool. Woops. So... So and then if we look at funcStorer now, it's undefined. Oh, that's weird. Oh, I changed it. So I guess I erased that return true. So it's undefined because this function is returning undefined. But if we return true, return true, and then did it. And we looked at funcStorer, and now holds the value true, and you can't, true is not a function. True is a value, so you couldn't call true, that doesn't make sense. So that's why we want funcStorer to be a function and not the result of the function. So that we have like this little window into that parent scope that we can access later, and it can be very powerful. And now, since funcStorer is a function, we can go ahead and call it. And we can call it again, and this is our counter. We can count.

Closure Questions

So, and similarly to our last exercise, we're just calling alerter later. So the setTimeout is Javascript calling it later, in one second, and here we are deciding to call it later, explicitly. Thumbs? It's okay if you have a middle thumb. Most people don't understand this. Thumbs online? Any questions? What is most confusing about this idea right now? So, I get the part where you're storing closureAlert in the funcStorer and that becomes a function, so there's like that permanence with X. Mhm? Because you're not really calling anything? Mhm. But I still don't understand, why is that called closure, or what's the closure in it? Yeah, so this part is the closure. The closure is the function that you can call later. And it's called a closure because it holds, it has reference to a variable from the environment from its parent environment. So this is a closure, funcStorer is a closure because it holds reference to X. So X is initialized and manipulated in a different scope. So we could do a bunch of stuff to X in this case we're just adding to it, but you could imagine that you could do a lot of different things. And since funcStorer holds reference to the environment, it's a closure. If you just wrap all of that code in an anonymous function with no name, is that, all that code would be in a closure as a function? Is that true, I guess? No, I'm not sure. So you're saying if I wrap all of this in-- Yeah, 'cause what I'm saying is you can either assign a function to a variable, assign, if you took off the one that says var funcStorer equals closureAlert. Mhm. If you took the parentheses off, you would just be reassigning the function to funcStorer, and it wouldn't be a closure, but if you leave the parentheses on, it is a closure. So really, it's just a function-- It's because, the key here about why we're calling it is because it's storing a function. So this closureAlert is returning a function. So what's the difference between var funcStorer equals closureAlert without parentheses and that one? Sure, that's a great question. So let's do that. So we can first just... Woops. So there's our closureAlert, right? If we just type out closureAlert, it's just a function that's not called, so it just prints out the function body. So if we called, we can call it funcStorer, this is where you store the func, and we just assign that to closureAlert like this. It's just gonna sign the function body of closureAlert. So now if we look at funckStore with a K, it's just a function body. And we could call it. What is this gonna return? A new alert function? Mhm. So it might, it would return something that looks like this, right? Right? Woops. And what we want, is we want to be able to call the alerter function from funcStorer. So what we could do... So this is myAlertFunc equals closureAlert, and so myAlertFunc is actually that function, which then you can call and it will actually alert. Versus, what did I call it? funcStorer. If I call it, it's just returning that function. We'd have to do something like this for it to work, and that's just weird. And it wouldn't work if we wanted to do it twice. It would always just alert once. So by storing that alerter function in a variable and being able to call it more than once, we can continue to have access to that parent scope, to this variable X. So it's not like you have to have a global variable, or you know what I mean? I'm trying to think of real-world applications, very simple real-world application for this, as opposed to saying I'm going to set X as a global variable in my window, and I'm messing with that. Mhm. It's a compact way to kind of still have that functionality. Mhm. Of keeping it local. Yeah, totally. So if we wanted to, let's see. So, another example, here, unless we have questions on this example? Yeah. The alert plus plus X, why does that increment X? That's just a shorthand. You can think of it saying X-- Not the alert, it gives you an alert box, right? Yeah, it would be in the alert. So it's gonna increment it, and then it will alert it, and that creates an alert box. That's just a browser feature. After that, do you see a console log like you would see it would be one? Yeah, right here? Yes. Yeah. And so, when you're saving it down the log, Mhm. Let's see if I understand this. When you're saving not only the function, but all the values of the function after the function right? Yeah, so you're saving all the values within the function, and also all the values it has access to in the scope. So also it has access to this parent scope, so it has access to its own scope, as well as previous scope. Which is the same as how we were doing it here, right? We just alert X, and this alerter function had access to X up here. And then for this example we had the settimeout, even though Alerter's being called later in one second, it still holds access to its parent scope. So the only difference for this one is that we're actually, we're just returning it. And then once we return in, then we can call it later. So for the settimeout example, Javascript's calling it later and this example we're calling it later. We're calling it later here. One more question. Can you do some of the whys? Why? Why you would do this? What's in it for the programmer? What's in it for the programmer? Hmm, there's a lot. Just give me the top three. The top three? The three-- Well, I'm gonna go through different use cases next. But I mean, the biggest thing is privacy, right? Is that you can make like an interface into, when we get into the module pattern I'll show you how you can kind of, you can make classes, basically classes in Javascript using closure, and you can limit how your program can interface with certain values. So once it's private, the only thing that can access it once you have this closure scope here, the only thing that can access and manipulate that data up here, is this function. And so you can really limit on how you can manipulate whatever variables you have in the closure. So that's, that's the biggest thing. But there's a lot of different use cases for it. Okay. Mhm. There was one more question here. Sure. The funcStorer calls alerter, she's asking if that's correct. Yes. Okay. It doesn't call closureAlerter. No, it does not call closureAlerter. 'Cause what happens is once we call closure alert, whatever that function returns is what's actually being stored in the variable. It's not actually, when we refer to funcStorer later, it's not looking up and calling closureAlerter again, closure alert at this point is being called once. And then when we reference it, down here, it's just referencing whatever this had returned from the beginning. So we call it again, it's no recalling closure alert. Beause if that was the case, then we'd be resetting X to zero every time.

Closures and Functional Programming

So here is an example of how we could use closure. So maybe we have this add function, and we want it to just make a generic add function, called like add5 = add 5. And now we have this add5 function that we can call later with another number, let's do a smaller number, and that would give us seven. Now let's walk through exactly how this is working. Let's play the game. Where did we leave off for our game? Did we leave off, Grace you went, Joe went, I think it was your turn. Let's walk through how this is running the browser. Well, you said the function add has a variable. Mhm. You skip down-- To the bottom? Cool. Perfect. Kim? Variable add five is add. So we call the add function, and then, who's up back there? William? So you call that function that the variable add is pointing to, so the value of five goes into the num. Mhm, so when we get into the function body num is now five. And then, Andy? Then we're creating a local variable called num1, setting it to the parameter five. Mhm. Kevin. And then new function, add two to that one. Mhm. And then you return that function without running it? Yeah. So we skip over the function body here, this never is run, then we just return, add to num1, and then Jake? Then it's complete. Almost. So return it, where does it return to? Add five. Yeah. So now var add five is whatever that function returns, which add to num1, right? So now this is a function, and then, Tanner? Then we're passing five, or wait. Passing five into the number? Well we just did that one, so we skip past that. Okay, an then we add two to five. Mhm. So we're calling the add5 function. Twice. With the argument two. Okay. Now, we only called it once. We called it here. This is a different one, this is add, now this one's add5. add5, remember, is gonna be this function here, 'cause that's what we returned. Okay. And then we pass it two. And then where does it go, Phil? Well, because add5 contains the contents, it also contains the variable num one (mumbles) scope. Mhm. Because it's five-- Mhm, so we hop right back up here, right? But you turn statement that it is, I mean two five? Yep, exactly. So we call add5 with two, we jump up here to the function body, num2 is now two, and then we add num1 plus num2. num1, again we got from the first time that we called add, right so num1 equals num, we got five from here, and because of closure, we have access to the environment here. The previous environment. The previous environment, yes. And that's how we got seven. So if you get add5 (mumbles) Right? Well, actually, we add five and do three, it's gonna be eight. Why? Beause the add parentheses five is retained. Mhm. As num1 equals five. Mhm. Because, even though we have reference to this scope here, every time we call add5, we're creating a new scope here. So num2 is gonna change. And we're not actually affecting num1. So num1 is being saved, num1 is always gonna be five in add5, but num2 is gonna be the one that changes. So when we pass two here, num2 is then two, and when we pass three at that point we make a new scope, this new scope here. But just because we have a new scope for the add to num1 function, that doesn't mean the parent scope is different, it's still the same. And the num2 would be three, which would be eight. So what if we didn't have that add to num function? Let's just assume that the whole function definition of add to num1 is gone. Mhm? I mean, I guess that defeats the purpose of having closure, but. Mhm. But you could still, so then that's just, then when you assign add parentheses five to add5, it's like-- It would call this whole thing over again. Yeah. And it would reset this variable. Right. So it would just be like a regular storing of function into variable, like your plain vanilla? Yep. Got it. Yeah. So the key here is that for the closures that we have a function in here, and then we're returning it so that we can call it later. Right. And then using the parent variable for permanence or-- Yeah. And actually, we don't need this here. This is just for clarification. Like, you could actually delete this and call this num. It would actually still work. So, just for fun, if you had, after that last statement, if you had add parentheses 10, and then you, I'm sorry could you add-- Here? No, at the very bottom. Mhm? If you said add5 equals add parentheses 10... add5 equals add 10? Yes. Mhm. And then you said add5 parenthesis two, what would you do? So this would be rewriting add5. So add5 was add parentheses five, right? Which is gonna be this function. This would overwrite, woah. This would overwrite it and make it 10. Okay. Doesn't like it when I click there. So if we did add5 two, then that would be 12. 'Cause we're just, whenever we use an assignment like this it's gonna completely override and just delete-- Whatever was there. Whatever was there, yeah. 'Cause f5 keeping that same reference in memory. No, it will overwrite that. Overwrite that, yeah. It will overwrite everything. But it would be better if we could just call that add10, right? But if we did want to do that, it would still be the same.

Closure Objects

So the other cool thing is that this also works if we're returning an object with methods. So the same rules apply, beause see look. We have these functions that are still inside this parent scope, even though we're actually returning it in an object. So let's explore what that looks like. Well, I'll type it out here, and I'll paste it. So we're gonna call... So here's like a counter, we're gonna call counter. Okay. So where did we leave off? Actually, I think, Phil, you were the last one, right last time? So we should start at the beginning. I guess that'll be you, Rich. Okay, so... It's like a global function, I guess, called counter. Mhm. It's basically initializes and then it finds that and then it goes down to the, skips the whole bottom So we said great, there's a counter, we skip it, and then Grace. We assigned myCounter, the return value of counter function. Sure, so we called counter and then it goes to the counter body, and then Joe? Sets the local variable N to zero. Mhm. Then myCounter is equal to the return value. Mhm. So you return that. What is this? The object of two functions. Yeah, so this is an object literal, it has two properties on it, one is count and one is reset, and they're both functions. So you can call it a method. It might not look like an object in this form, but if we delete this, you can see that there's just an object with our properties in there. So we're returning an object, great. So what is myCounter now, Kim? Object count colon the function. Mhm, totally. And then if we wanted to increment the counter, how would we do that, William? Knowing that myCounter is an object. Actually, let's inspect it before we jump ahead. So we have myCounter is an object with two properties. So if we wanted to count, we wanted our counter to count, how do we access the counter property? Dot? Yeah, we could use dot. Dot what? Dot count. Dot count, like this? Up? So if we don't call it right, it's just gonna be the name, so we need to invoke our function, and it's gonna increment it. So there's one, and then we can keep doing it, right, it's just gonna keep counting. So we can see myCounter, still just an object, we haven't affected anything, and then if we wanted to reset it, could we do this? Yep, that will also work. We can use bracket notation. And so... Count like that. So as we see, we can return the functions from inside of an object as well. And this N is still holding access to its parent scope in its previous environment. So if we wanted to call it again, oops. That's myCounter.count, and we have to call that since it's a function. It's gonna be one. That's gonna be two, et cetera. Cool? Thumbs? The only difference between this and the previous example is that we're using the dot notation to access the function in the object. Yeah. That's being returned from the-- Right. so the main difference is that for this one we're just returning one function, and in the next one we're returning an object that contains two functions. So if you wanted to have to do different things like reset the counter, you're gonna need two different functions, right? So this is how you would do that.

Closure Recipe

And here's just like a recipe, sort of outline, sort of the ideas of like, how do we create a closure? So the first thing is you have to create the parent function so here we have the functions checks go, and it's a picture so I can't highlight it. It goes here. Then we define some variables in the parent's local scope up here, this inner var, and then you put a function inside of it, so the function again could be inside and object as well, but it has to be a function. And then you return that function, without calling it, from the parent function. Right, so that's how you create the closure, or you start to get there, and then if you execute, this is how you would execute it. So here's our check scope, right? We create a variable, and we store that return function from check scope, so we call check scope here, it goes inside the bracket, we create a local variable, then we see, oh great, there's an inner func, we skip the inner func and then we return it here. So we return this inner func, so test is now inner func, which we then just return scope. And then if you just check test and you make sure it's a function, this is optional, and then you run test and test will return if it's local scope or global scope, and of course we know, since we're all experts at scope now, that it's gonna return local scope. Cool? So that's just like a recipe and a way to execute for when you're doing the exercises, you can refer back to these slides to sort of get a step-by-step. Cool. So-- There's a slide called "Gotcha!" (laughs) It's not that bad, it's not that bad. So we have this sayAlice function, and we're gonna call it. And let's just... And let's just see what it says. So let's start with, who did we leave off? Kevin, did you go last time? Okay, so we'll start with Kevin. So define the function called sayAlice. Yep, so we define a function called sayAlice, and then Ben? Skips to the-- You know, this class has been the most picked up the skipping the function body, faster than any other class I've ever taught. Good job, guys. Go Minnesota. Yeah, it's Minnesota, that's what it is. The online students, too. Yeah, and the world. Maybe I just got better at really emphasizing that. I don't know. So we skip it, and then, Jake? So I guess we start at the bottom and it's variable of what, sayAlice. Mhm, so we call sayAlice, and then Tanner? We go into our sayAlice function? Yeah, we go into the function body. And then, Phil? So the first thing you do is you skip makeLog. Mhm. Well, we recognize that there is a makeLog in storer, and then we skip it. And then you assign the ticks to alice. Mhm. And then, John? So we're here. Then sort of return to function makeLog back to the call. Mhm. So it's returning this function, so what is now a function. And if we call, oh I know which is. What does she say? That's what we should call it. What does she say? Oops. So what is it gonna... So what it is it going to console.log? Undefined? That's what we get? It's actually going to console log "Why hello there, Alice!" And the reason is is because, if you notice when we walk through this colored line-by-line, we initialize this variable Alice, and it exists. By the time that we call it down here, Alice is already been created and assigned a value. So does it matter, so it's still, because it'll still a parent to makeLog, so makeLog has Alice available whether it's done above or below? Right. Unless, like, you did-- Right, then that's-- If you move this up here. Right. Then it wouldn't work. Or something. Yep? I have a more fundamental question. Sure. On the very first line where you see var sayAlice equals function. Mhm? Could you rewrite that to say function space sayAlice parentheses? Yes, you could. And that would be the same thing? Basically, yes. Mhm. The cool thing about this, the cool thing about the fact that we can save a function to a variable like this shows us that in Javascript, a function is just data. So that's kind of cool. And that's why we can return functions from functions, like makeLog, is because of that. So if on the second line makeLog, if you wrote that function space makeLog parentheses, if you wrote it that way you can still return makeLog? Yeah, you can still you can return it any way. So one is having, so this is an anonymous function stored in a variable. If we had makeLog like this, function makeLog, this is called a named function. So this is-- Where it can in this case the same. Yeah, it'll work the same way. You can-- So why would you do one over the other? You can even do two different ones. Why would you do one over the other? Is it a matter of personal preference, or? It's mostly so, it has a lot to do with scope, actually, is where the, at what point in your code can you call the function. If you store it in a variable like makeLog, then you can, you can't call the function until this line has been called in that scope. But if you had it the other way, we could call makeLog up here. Something called hoisting, which happens to all variables, but when you name a function, it doesn't hoist the initialization of it, it hoists the entire definition, so you could call it, and the reason it's a little bit dangerous to do it this way is because if you have like an if-L statement with two functions of the same name, and the hoisting will overwrite it in ways that aren't expected. So in general I recommend just to to call it, to store in a variable like this. Yep. Cool. So, I think I was confused about the value of alice because we're defining alice after makeLog. Mhm? But because it's still in the parent scope that console log has access to variable? Mhm, because we actually initialized this, and, excuse me, and gave it a value before we actually called makeLog. Is that because of hoisting as well? It hoists it to the top of the function? The sayAlice function? Yeah, that's part of it. But the fact that it has a value is because of the order in which it's called. Yeah. So here's a stopwatch example. So, let's see. So makeStopwatch is called here, so we enter into this block of code, and it's gonna console.log initialized, and we're gonna set elapsed at zero, then we're gonna console log elapsed just to see what it is. And you'll notice too that these two console logs will only be run once every time you call makeStopwatch. So even if we called X down here, these two console logs won't run, because we're not ever going back up here. We're just gonna go immediately inside of this function. So then we have the stopwatch function, which all we'll do is console log stopwatch, and return elapsed. But we're gonna skip over it because we haven't even called it, then we have this increase function, which is on a set interval. So like a set interval, sorry like settimeout that we saw earlier, so interval will just, it's not a blocking operation, it will just call increase every one second basically forever. So it will just keep calling increase. So this is our stopwatch, it will keep the time. And then we return stopwatch, and again, stopwatch is this function, that's just gonna return elapsed. So let's investigate. So let's investigate what happens. Let's pop this out, make it, see if I can make it bigger. No. Hm. It's not making it, oh there we go. So I'm just putting that code in there, and then we can say var watch equals makeStopwatch, and we're gonna call it. And so we see that initialize and zero are called. If we have otherWatch, we can do makeStopwatch again, woops, makeStopwatch again, and we see that those two console logs happen at the very beginning here again, however, when we call watch, initialize and zero aren't running. And that shows us that this code up here is only run once, so that scope is created once, and then when we call this down here, the body in here will be run again. So we have watch, and then we have otherWatch, 38, ect. Does that make sense? So that's just another example about how you can use closure.

Closure Exercise Solution

So the first one was write a function called nonsense that takes an input string, this function contains another function, blab, which alerts that string and is immediately in call, is immediately called inside the function nonsense. And here's an example of what blab should look like, just a function, it alerts string, and so we have our function, nonsense, here, I called the parameter string, and then if we were gonna call nonsense we could say nonsense, and we can, oh actually I already did it down there. So we can call nonsense down here with blah blah blah, so the string is now this blah blah blah, and then we enter through the function body here. We see that there's another function, we skip it, we call blab, and blab then alerts what? Yep, blah blah blah, exactly. So for the second one, we're doing something a little bit different. Instead of immediately calling blab, we're calling it after two seconds, so we're settimeout, and then it will still alert blab. So, for the third one instead of calling it inside or doing a settimeout, we're just gonna return the function body. So we see here on line 34, we return blab, which is gonna return the function body. So blab later is actually gonna be the function body here and it's gonna alert when we call it. So if we call blab later here it's going to what? What is it going to alert? Kevin, do you know? (whispering) Blah blah blah. Yeah, exactly. So it's gonna alert blah blah blah. (whispering) And the blab again later. We called it. What's that gonna return? Tanner? It's gonna return blah blah blah? Close. It's actually gonna return hee hee hee. Because we called nonsense with hee hee hee, so string in this case is gonna be hee hee hee. Can you guys see that in the back? Cool? What if we did blab later here? What's gonna alert? John. Could you repeat the question, please? Sure, oh I'm sorry I didn't realize you were on your phone. If we called blab again later in here, what is that gonna return? What is this gonna alert? Blah blah blah. Yep, so it's still gonna alert blah blah blah, so the fact that blab again later is alerting hee hee hee doesn't affect the blah blah blah. If that makes any sense. (laughs) And why is that? Why are they two separate things? Grace, do you remember? Oh, Tanner remembers. Why, Tanner. I just want redemption because you're passing me the variables, so after nonsense. Yeah, absolutely. And every time we call that function, the inner function's creating a new scope. Even though they both are sharing the same parent scope, it's new. It's a new inner scope here for this function. Cool. Dun dun dun. So this is a function called lastNameTrier, you pass it your first name and it creates this complicated algorithm that can, you can try on last names. Like little girls do when they have a crush, you know. If you have lots of crushes you can see which one sounds better with your name. So how it works is you pass in your first name, so if your first name is Farmer you would pass Farmer, and then you could try on the last name Brown, that's gonna log Farmer Brown. And it's also really convenient if like you really, really like someone and you wanna just write their name all over everything, you can just like do this a bunch of times. You know? I don't think boys do this, but girls they write their crushes' names a lot. No? No, no one? Okay. Just me, all right ,fine. Do you use gel pens? What? Yeah, with gel pens, absolutely. you gotta be careful 'cause they smear, you know? There's an art to it. Okay, anyway, too much information. So here if you wanna try on different farmer names, so Farmer Jane, Farmer Lynn, for example. That's for the farmer to find out if he's gonna marry the Jane or the Lynn. Yeah. So guys could use this function. Yeah, absolutely. It's a gender-neutral. Or it could be a female farmer, too. Yeah, mhm. Marry a farmer. Or if you're giving birth and you know that your prodigy is gonna be a farmer, you can match that, so you know, it's-- It's very unisex, yeah. Yeah. Cool. Anyway. So here we have our function, so our parent function is here, and then we have an inner function here. And again we're returning that inner. So how we'd run this is, we'd say, oh and I forgot to mention, Katrina is our, these are Katrina's solutions. So she's awesome. Look her up, add her on Twitter, all those things. So Katrina, so var first name Katrina, pass the string Katrina and then to lastNameTrier, so the string Katrina is now a first name, we have this inner function, we skip it, and we return in. so first name Katrina now is the value of this inner function. So now when we call first name Katrina we can pass her last name, which I'm not gonna try to pronounce, Wychaco, I guess I tried it. So Wychaco is last name, and then we're just going to console.log the first name plus the last name. The cool thing here is that this parameter stays the same, Katrina, we don't actually have to say var first name like this, oops, equals first name or anything like that, it just holds reference to the parameters, 'cause it also live in the scope. So we can just do it like that, and then you can also you know, try it with Smith as well. Any questions on that? So it's like, you can do like Fibonacci series with these because they're storing the results of the next... Yeah. Yeah, you can do, you can do a lot of different things with it. I think there's one thing, oh there's one thing I wanted to mention for this nonsense function, a common mistake that I see is that when you are writing the inner function, you will also give it a parameter name of the same name from the parent function, and that's not gonna work because if you don't pass anything to blab, so when we're calling blab here with no arguments, that means string is now undefined, and the most local, the most local variable value is the one that wins. So this alert here is gonna then alert undefined because it's being overwritten by this parameter name. So if you're seeing that problem, just double-check your inner functions parameter and make sure that that function is actually taking an argument. if it's not, then it's gonna be undefined. Cool. so we have lastNameTrier and then we have this storyWriter. So this is a function that returns an object with two methods. One method is addWords, which will add words to your function, and the other one is erase. So you might imagine that this is like that game that people play where you start off with a story, you start off with like the beginning of a sentence, then the next person adds a sentence, and then you go on and on. You create a big story. Has anyone played that like on road trips before? Not in Minnesota, okay some of you have, okay. So we have, just an example of how we'd use it, we have this variable, farmLoveStory, and we have storyWriter, which is function that we call, and then farmLoveStory is now gonna be an object that has a property addWords, which you can then pass a string. And so the string was, "There was once a lonely cow." And that will just return "There was once a lonely cow," and then you can add more words, and it will concatenate that string and say "There was once a lonely cow "that saw a friendly face," because we just added this there and then another example would be like storyOfMyLife, which is "My code broke" and then "I ate some ice cream." It's pretty consistent in my day-to-day. And then you can erase it. So that's how you would implement. That would be the functionality of the storyWriter function. And so this is what it looks like on the inside. So in a closure scope we have our story, which just starts off empty, right? And then we return this object, and this object has two properties, the first one's addWords, which is a method, and this method takes a string, and story is you add the string, and then I put a space in between, and then I return the story.trim, and then what that does is it just it doesn't show, it deletes the last, the trailing space, so that it wouldn't look, so it doesn't print like "I ate some ice cream" and then space. It doesn't actually trim off, it just shows a copy of the string without the last space, so the space actually still exists in story. And the other, and the erase function's pretty simple, you just set story back to an empty string. Cool? Any questions about that? I wasn't getting, so when you run that, where do you see that on the, that'll actually print that out on the console? 'Cause I'm-- Yeah, that's a good-- Wasn't get defined, you know? In the console? Well, not in the console, in the... What do you mean? Like this one? It builds it correctly but it doesn't operate to the console like your examples show us it should. Mm. Are you returning? Are you saying my example isn't? No, mine isn't. Okay, make sure that you're returning. Mine'll trim, but I just return (mumbles) Mine doesn't either. Since it's not being executed from the console it's not gonna dump it all here unless you actually have a console log statement? To execute the call from a console statement. Mm. 'Cause if it has the return value, then it would show it. Right. So refreshing the page it doesn't necessarily-- Are you doing it in JS Bin? I was not. None of us were using the strings JS file from yesterday. Okay. Yeah, you can just run it directly in your console, like this and it should print out. It's doing it now. Cool.

Callbacks

Module Pattern

(music) So the next thing that I want to talk about just at a really high level is the module pattern. The module pattern, there's a little more to it, but in in essence, the module pattern is a way that you can modulize your code or you can do things like make sort of, class-like structures in JavaScript. It's a way to construct sort of like an API, which is just like an interface. Of interacting with your data. And so for example, you can have some private methods and private properties. And then this returns an object that has some public properties which can be accessed by the public. And there's public methods that can be run. And then there's a privileged method that could access the private method. And what this does, is this limits what your application can do with the data inside of here. And this is just sort of a overview. And we'll go into more detail of how we'd actually use it. And so here's an example of a Car function. It's a constructor function. We can create different kinds of cars like this. This would be like if you had a car game, and you didn't want people to cheat and change the gasoline level, like maybe this is like the most boring car game ever. Where you keep going until you run out of gas. And so, how you would what you would do to sort of limit that cheating is that we'd have this privileged method here that's called go. Which calls useGas, and passes the speed. And so what this does is that you can't change the gasoline level directly on the car. And let's just run this, just to kind of look at it, so. Oh no, my timer's still going. So we have here our car. So we can say like var, what's a kind of car? I don't know. Ferrari. Ferrari. Fer, how do you spell Ferrari? F-e-r-r-a-r-i. Okay, equals car. Does it take anything? No. So now we have our Ferrari, and we can look at it. So our Ferrari is an object that has a go function. You can change the radio station. And also you can just see the radio station. And so, if we wanted to say ferrari dot go, and we can pass our speed. So let's say we're going two... Everyone goes, the speed limit's about two, right? Naw, okay. So quiet today you guys. And so you can keep going until you run out of gas. And it will print out of gas. And so we see here, when we useGas, we pass it speed, and it goes up here. And useGas is a private function. So we can't change how the gas is used. And so this would prevent someone from cheating on the car game. Where they would give themselves more gas than someone else for example. And the whole idea here is sort of abstracting away what is important and what is not dangerous for someone to use. If you think about just the car metaphor in general. When we talk about APIs, we would say like, to drive a car, you don't have to know how the engine works, right? And in fact, it's better that when you're driving the car that you're not touching the engine. You're just only touching what is public to you. Which is the steering wheel, maybe the ignition. The shifter thing, you know, and... So our car API is abstracging away how the engine's actually working. So, you can drive a car and not actually understand how the engine works. And it might be dangerous if you did and you went in there and started messing around with it, and then it broke. And so that's the same idea here. So we have some things that are private in our car, like the gasoline level. And how to use the gas in general. Whereas, something that's less dangerous, like changing the radio station, or just going. So go would be like pressing the ignition. We have access to that. And so we can use closures to create an interface like that. That make sense? Thumbs on that. Thumbs. Cool. So is this, kind of building on the closure? So this is using a closure. See how we're returning this object? The use gas is kind of the function within the function? Yeah. Exactly. Exactly. And that's just another way to think about closure. Any questions about closures? Because we're going to keep moving. We're good? Make a comment on that, when before, when we were naming a function it was at global scope. This one is not. It's at the function scope, because it's defined within that function. Yeah, exactly. And you can use the module pattern to organize your whole code base into different modules as well. And so you have like, private components of your app that you can use. So the module pattern is used all over. And this is just a simple example of, like how we could use the module pattern to make classes. As like a factory to make objects. Hmhm? Can you call a function within one of the functions that you're returning? So, for example, here could you call the go function within the changeStation function? Yeah, you could. Okay.

Higher-order Functions & Callbacks

So now we're going to talk about higher-order functions and callbacks. So what is a higher-order function? So a higher-order function either takes a function as an argument, or it returns a function as an output. So, we've done a lot of returning functions when we were going through closures. Now we're going to talk about how functions take functions. And how to master that. So then we can, once we get to deep diving into our functional programming stuff, we will like have those concepts mastered, and they won't be confusing hopefully. So here's an example of a function taking a function as an argument. So this is adding a click event. And then the callback would be this function here. So a callback is a function as an argument. And then here this looks familiar to us, because we've just been doing it. And this is returning a function from inside a function. Cool. So, callbacks. So here's an example of an ifElse statement abstracted into a function. And so in functional programming, we're using functions rather than procedures I guess. Or we're using we're using functions as units of abstraction. Versus abstracting over data, in functional programming. And so, some silly example is you could just use an ifElse function, instead of an actual ifElse statement. And so here's an example of that. Where we are passing a condition. And then we're passing a callback function for the true condition. And then we're passing a another callback for the isFalse condition. So, here for our example, we are just passing true. So the condition is true and it's going to call, or it's going to enter into this block. And so what's going to happen when this code runs? Console log true? It's going to print out the function? The content of the function. All very good guesses. So what's actually going to happen is it's going to return undefined. So we're not returning. We're not returning these things. And we're not calling them. Oops. So it's important to note that when you are doing this you need to call it. And so when we call a function that's a parameter or that's an argument, it's just like referencing it as a variable. So, if we wanted to call isFalse, we just add the invocation operator, also known as two parentheses, next to that function name. And so this is, these are two anonymous functions that we're passing in. But we could also pass in other variable names. So if we have this function stored at logTrue, we could pass it than as logTrue or logFalse. And that would work just the same. IsTrue would still be, would then be logTrue, would be this function. And isFalse would then be logFalse. And so if we we did this, what would happen? What's going to happen? Kim do you know? What is it? Log false? Yep, it's just going to log false. Exactly. You go to the previous slide. If so, now, silly question. You can't put the invocations in the in the parameter. No, so. That's just bad syntax, or whatever. Yeah, it would just give you an error. So, the thing about parameters, they're just like uninitialized variables until you pass an argument. So just like a variable, you couldn't call. Like, if we just had like var x here, you know it's just an uninitialized variable, you can't call x, because x is just undefined. And so it wouldn't make sense. Like we're just here declaring declaring them as like undefined. Just saying like if there is an argument passed, in this place, it's going to hold a value. If not, it's going to be undefined. Hmhm. There's a question on the chat room. Can you walk through this step by step. They're stuck on the ifElse function and how it works. Oh, yeah, sure. So here, let's do it in the next slide. So, how our browser would read this. It would say, okay, great, we're creating a space in memory called ifElse. We're going to store a function there. This function just happens to have three parameters by these names. Which at this point, they have no value, right. Then we say logTrue is another function. LogFalse is another function. So then we pass to our ifElse false, logTrue and logFalse. So once we call the function, we're going to enter into the function body. And at this point is when our parameters get their values. So, condition is going to be false. IsTrue is going to be logTrue. IsFalse is then logFalse. So we go into the function body now and it says if condition, and so what happens when you put a variable or a an expression in between these two parentheses, it's going to force it into a true or false value, into a boolean value. And so, luckily, ours is already a boolean value. So condition is false. So if it's false, it's not going to enter into this block. It will enter into the else block. However, if this was true, so if we changed this down here to true. So, if condition, so this is the same as saying if condition is true if the value at condition, stored in condition, is true, enter into this part. And then, since we're in this part of the block, this if block, it's going to call isTrue. Otherwise, if it's false, it's going to do the else. And then once it calls isTrue, it just skips to the bottom here. And then basically, you can just imagine, it says return undefined. So, if it's true, it goes here, and then it skips down there. Which means that this else block is never entered if this condition is true. If this condition is false then this is never run. So and then you can read this function as like like if true, do this, else do this.

Passing Arguments

So we passed in callbacks. We called them, like but what if we want to add an argument? So here's an example of us passing arguments to our function. So we have a couple math functions here. One is called increment. One is called square. So increment just adds one. Square is going to multiply itself, it's going to multiply a number by itself. And here's our function called doMathSoIDontHaveTo. And what it does is it takes a number, and then it takes a function. And what it does is it applies, it calls that function. With that number. And so we implement this by just passing it five, so that's in, and then square, which is the function up here. We enter into the function body. And it's just going to return it's going to return square with five, which is what? Ooo. What is this? This is a math test. Twenty-five. And what about this one. Five. Five. So this is just an example of, we just call the function as if it's just stored in a variable name. There's no difference here. So how would we pass an argument here, to either isTrue or isFalse? What would have to change? To pass a argument? Put it in the function within the if block? Uh-huh. The first thing we'd have to do like this, right? Assuming that we're getting our argument as being passed in from the parent function, we'd have to put it here. And then ya, we'd just pass it just like this. Or we could just say, even like this. We could say var arg equals zero. Or something, and then we can just have our arg inside the function. And we could do it like that. Either way. And then how would we, how would we call this function? Pull ifElse. And then put your condition, your true isFalse, and then your arguments. Mmhmm. And this would be a function. This would be a function. And then pass a number, or whatever. Maybe a string. Cool. Now. Sorry. Mmhmm. Because when we first did some of, when you did the a plus b plus c yesterday and it was a return a plus b. Because it didn't care if you had three arguments. Mmhmm. Even if you don't specify it there in the function definition, the arg, it's just. So, if you don't have that arg up top there in the function definition, and but you ran ifElse, and you had that asdf at the end, it's just going to like, not going to do anything with that. It's just seemed like the c. Exactly. Exactly. Can you explain why we have to define the arg variable? Like why couldn't we just when we actually called it as true function, just pass in the parameter at the time of calling it. Mmm. Because, like here you mean? So if we had this function here. Like here? Right. And then when you Go through the asdf in the-- Exactly. parentheses, in function indications. Mmm. Okay. Because this here, this is just a function definition. We're not invoking the function here. So the only way that we can give a parameter a value or pass an argument is when you're actually calling the function. So, we're actually calling the function at this line. We're not calling the function here. We're just passing the definition. So, if we were to call the function, you have to have the two parentheses. So how does the, like if you have a click event, in JQuery, I'm just thinking. Can you pass the event variable in the callback function? How does that work? That's being. So, whenever that element is clicked, JQuery is calling your callback function with that event object. So like when you. So I have an example in the very top. So, here or sometimes you call it e. You've probably seen it as like e dot preventDefault. Right? Or something, or something like that. Here you're only defining e as a parameter. So you're just saying I'm just going to call this e. But you could call it, you know, this asdf, and do it that way. So, whenever, say we had a click event on like just the body here. We click on it and at that moment, that function is called. So, this function here is called at the moment of clicking and that's why we get data for the e. So like the e object, or the event object, has data on like where you actually clicked. And stuff, and that's, so it's being passed at the time of clicking. Mmhmm? The question is how does arg get passed to isTrue and isFalse if the parameter is only set on ifElse? Let's see. One more time. So how does that arg get passed to isTrue and isFalse if it's only set on the ifElse? The function definition would have to take the parameter then use that parameter in its definition. Or you'd have to use the arguments function to, or the arguments semi-array to get access to it. So yeah. Maybe this will answer the question. So those are just example functions. But this function then is a function that takes that argument and then passes it to the function. And it will just console dot log here, x. So x is a parameter. So this is where a parameter and arguments get a little bit tricky. So, arg, at this point, is a parameter. X here is a parameter. Asdf, right, is a value that is an argument. All of these things are arguments in here. So, true is an argument. This function definition is an argument. And when we pass asdf, arg now contains that value. Arg here is now, you can imagine it's asdf. And that's an argument. And now isTrue, this x here, and isTrue is then going to be asdf. And that's how you follow it through the function. Follow up question to that, I think. Do you have to set that parameter in the function definition of isTrue and isFalse? No, you don't. So the other option is if you have no parameters, or you don't know how many parameters, you can use the arguments keyword. Which is going to print out an array of the arguments. So if we only pass one argument, and it's asdf, it'll have asdf, in an array-like object. So we covered this yesterday in the function section, if you want to review the arguments keyword. It's pretty interesting. Cool.

Callbacks Exercise

So I have just a short exercise for this. We're going to go back to the Functional JS Repo. And it's in the callback section. So we have this callback. Only a few, but we also have one last thing for the module pattern enclosures. That number seven, the toaster one. If you want to design your toaster. You can get started with that. So what we're doing is in the closures we're doing number seven, which is the toaster. And then for callbacks we're going to do all of them all of them one through three. And then there's also extra credit if you finish early that combines callbacks with closures. It's pretty interesting.

Callbacks Exercise Solution

So we were just doing the last closure exercise. Which is to use the module pattern to create to create a toaster. And there's a bunch of different ways you can do this. This is just sort of like expressing I don't know, your creative side, when making your program. So this is an example of a toaster that has a max temp of 500. So how we'd call this, we'd say var, this is actually going to be toaster, so var myToaster equals Toaster. Call it. And then you can say, myToaster dot setTemp. I'm sorry, this should also be private. SetTemp equals setTemp equals, I don't know, like 300. And then so, I'm sorry we should pass it 300. Like this, since it's a function. So myToaster dot setTemp. We're going to pass it 300. So then this says if the new temp here, We want a comma after (murmur)? Ah, nope. Because, oh, good point. Whoa. There we go. Thank you. So we have setTemp, we're going to pass it 300. So newTemp is now 300. And it says if newTemp is greater than maxTemp. MaxTemp is 500. We'll console log That temp is too high. Otherwise, Temp equals newTemp. And so what this does is this prevents people from being able to set the temperature of your toaster to too high. Anyone else, how else did people implement this? Have a different implementation? Rich, I think you had one, what was yours? I just kind of modeled over the car one. And let the user, like increment the, you know like the light-dark type thing. Mmhmm. Okay, so like the timer kind of. Yeah, like exactly. And then they're going to input. I didn't finish the code, but they're going to input, like if it was just one slice then it wasn't going to go for as long. And if it was two slices it would, or whatever. But it wouldn't let the user mess with that inner workings of how long the toaster should run. So kind of similar to here. Like, your user can't change the max temp to override. Right, cool. Awesome, so that's, is an example of, really it's an example of making a class-like constructor in JavaScript. Which is cool. So in the second thing that we were going to do. That we did, was the callback exercises. So I'm just going to paste this in. Do this really quick. Alright. So the first one says write a function funcCaller that takes func, which would be a function, and an arg, which can be any kind of data, and the function returns func called with the arg as an argument. So the first thing we need to do is to write a function called funcCaller. So and that just equals a function. And again it takes a func. And then it takes an arg. And Excuse me. And then we are going to call func with arg and return it. So that's the first one. The second one is a function called firstVal. So we say firstVal equals a function. Right, there's our function body. And our first parameters is going to be an array. And the next one is going to be a function. And we're going to call the function with the first value of the array. So array at the zeroth index. Then we're going to call it with that index, which is going to be zero. Then we're going to pass the entire array. Cool? That's where I kind of got tripped up. Did you have a question about it? Yeah, I guess it's maybe the wording of the thing. So. It's going to call the function with just those three arguments. What happens if you get that first one, next one, I didn't understand index pound index number. Oh yeah, index number. And then the whole array here. Cool. And so now we're going to write a different version of firstVal that doesn't matter if it's an array or object. So we're just going to call it a list that's going to take a func. So what we can do here, we could say if and we can say array dot isArray. So this is just a method. That checks if something is an array. So, we can just pass, and we pass list to it, and it'll return true if it's an array. So, if it's an array, we're just going to do the same as before. Right, we're just going to pass the first one. Else Else we are going to deal with it if it's an object. So we're just assuming that's either an array or an object. So otherwise. So one way we could do this is we could just loop through so var i, I'm sorry, var k and list. So we can just loop through the list. And then we can just return out of it. Return. And this would just loop once. K, or list at k. So this is just a property value and this is the property. And this is the entire list. And if you return out of it then it will stop looping. And we can just return this one too, so it's consistent. So that's one way to do it. You can also just do... So there's this one method called object dot keys. And then you can pass the list there. And this returns an array of all of the properties. So we can call this propArray. And then we can say then we can say, func and then we don't need to return it. We'd say func propArray okay, func list, and this is getting the value list. PropArray at i Comma, propArray at i. Which again is the property name. So object dot keys, what it does is it returns an array of all the keys in that object. And then list. So that's just another way that you could do it. Where are the Is coming from? Because. Oh, I'm sorry, that should be zero. I'm just so used to doing i. Thank you. Yeah, zero.

Underscore.js

Underscore.js Introduction

(music) Alright, so now we're finally to the point where we can talk about Underscore, so we talked about scope, we talked about how closures work, we talk about the complexities between a parameter versus an argument and how all these components work together. And now, we can use a utility library like Underscore.js, which combines all of these concepts into a handy library that gives us all these utility functions that allow us to do some pretty cool things. So the first thing is, has anyone used a library here? Like, raise your hand if you've used a library before, cool. So, Underscore's a library which basically, a library, in a sense, is just a collection of methods available to you, so we have a jQuery library, which provides us a function, a jQuery function, that looks like this. Right? And we can call it like that. Underscore is a library that gives us an object that's an underscore with a bunch of different properties on it. So, the syntax for Underscore, you'll see an underscore and a dot, and you can just imagine that somewhere in the library it underscore, I'm sorry, it says var underscore equals some object. And that's why we use dot notation, and then we can say underscore dot, you know, some method equals some function and then, you know, some function that does something. So, that's like how it's working at its core, and we just include that JavaScript file in our HTML, we have access to all those methods. And let's just check out the Underscore library, just kind of get a feel of what it looks like. So, this is just their documentation. You can see that that Underscore comes with all of these handy functions that you can use that make your code easier. The cool thing about Underscore is it works in all environments, it's just pure JavaScript. It's not like jQuery where it only works in the browser, so you could use Underscore, for example, in Node it's used a lot, or if you're doing some hardware stuff as well. And then you can also look at the annotated source code, which is really interesting, you can see like how they actually write Underscore, and it's all annotated, so you could see like how Reduce is written, here. At the very bottom, yeah So that's really handy if you're getting stuck, and you kind of want to see how it's really working underneath, yep? Did you say jQuery only works in the browser? Yes, if you think about it, jQuery's all about selecting DOM Nodes and then manipulating them, and then putting them back into the DOM. The DOM is the browser, lives in the browser, cool. So we're going to talk about a couple jQuery methods, I'm sorry, Underscore methods, how to use them, and then I'm kind of going to set you loose and let you just explore and work with Underscore on some exercises.

\_.each()

So the first one that I want to talk about is \_.each. This is one of the most common Underscore methods that's used and it's not only used in functional programming in fact, a lot of people use this. I want to argue that most people use \_.each, either in Underscore's \_.each, jQuery has its own version of \_.each, and then there's a native for \_.each as well, that comes with JavaScript. So what \_.each does is that it takes a list, it could be an object or an array, and for every element in that array, it calls the callback function here. So, now you guys can see what I was talking about with the value, the index, and the lists, and from the exercises because that's how \_.each works. So, in this example, this function here, is going to be called three times. The first time is going to be called with one, as the value, And then zero and the whole array. The second time val is going to be two, and the index is going to be what? What's the index going to be? One. One, and then the whole list right, that array, and then the last time, the val is going to be three, i's going to be two, and again the list is going to be the whole list. And we know, from our par discussions, that we don't have to work with all the parameters, right? So, this in this signature here, this function signature, those are just parameter names. We don't have to reference all of them, so if we just wanted to console that log to val, that's fine, but if we want to do all of them, we could also do that too, and we also know that we could call this whatever we want, right? Sometimes it's called element, you can spell it all out, index, I don't know, you can call it whatever you want. And so, as long as they match in the function body, that's what's important. So, as far as the order of element, index, and the whole list, that order doesn't change, so it's always going to be the value, it's always going to be the index, and then it's always going to be the list, in that order. That's the order of the parameters, so whatever you call it, call those parameters, it's still going to be passed in that order, so for example, let me just go back. So we had it like this before, val, i, and list, right? If we swapped val and i, like that i would then be val, so you can't mix these up. You can name them whatever you want, but it's not going to affect the value that it holds, so this first one is always going to be the value, and the second one is always going to be the index, cool? And is the quick way to kind of, instead of having a function that loops through a collection. Yeah, and so this is awesome because you don't have to write out the loop the whole time, it's more obvious what it's doing where your attention in loop syntax, kind of is drawn to the mechanics of the loop, like where does it start, up until when, etcetera. And then also, \_.each will work with both arrays and objects, so you don't have to worry about the four n versus the four loop with the semicolons and then, or the length, or having to do has own property, and those kind of things you don't have to worry about that with \_.each and it kind of just takes care of it for you, which is great, however the native for \_.each, in JavaScript I think is only for arrays, so keep that in mind, but \_.each in Underscore can do both. Oops my email's open. Yeah so, awesome, so let's look at it a little more. So we're going to use \_.each with this array, Say my pocketmon array with the Charisaur, Bulbazard, and Twomew, and we have this function logger. All logger does is it takes a value, and it console.logs that value. And so, how are we going to use \_.each to log all the values of our pocketmon? So we start with an \_.each like this, let's see William, do you know? Your first parameter's the arrays of pocketmon Pocketman, yep, and then the second one? Do you know Grace? Logger? Logger, yeah. So iterator is just a fancy word for the function that's going to be called every time in the loop. So yeah, and that's basically how we use \_.each, is we pass it our array, and then each time the loop is called, it's going to call this logger function with, first it's going to say Charisaur, then it's going to be Bulbazard, and then it's going to be Twomew, so that's the order. Oh and I wanted to introduce, remember AnimalMaker, from yesterday we have this AnimalMaker constructor function that creates our animals, so we had some animal names, can someone give me some animal names? Smokey. Smokey. Fluffy. Fluffy. Trigger. Trigger. So, what we want to do here is the same as yesterday, we want to loop through all the animal names, create an animal object that has the speak method on it, and then add them to our farm. And so this is how we did it with just a regular loop. We did a four loop, we started at zero, and we looped while i < animalNames.length, and then we said i++, we incremented i each time by one. And then we are running each animal name, so Smokey, Fluffy, Trigger, one at a time into our AnimalMaker function so we see here that we're calling a function with the animal name at i, and then we're pushing that whole thing into our farm. And so, my challenge to you is to take a couple minutes and rewrite this using \_.each. Anyone know what the first argument we should pass to \_.each should be? Animal names? Mm-hmm. And the second one's going to be a function, right? Let's move this up so we have more space. So what are we going to put inside the function? Umm, farm push? So, farm.push, and what are we going to push? Andy, do you know what we are going to push? I think we're going to push name, right? Close, we do need to use the name. We need to make sure that we don't forget to pass it through our constructor function AnimalMaker. What about, Kim, do you have an idea of what we should add to this line? AnimalMaker? Mm-hmm. Like that? Yeah, cool. And so now we can just delete this, or actually let's take a moment and sort of compare how these look, right? You remember how I was saying that for our four loop, the attention is drawn more to the mechanics of the loop rather than what it's actually doing. You can see this pretty clearly here. The mechanics of the loop actually takes a lot of typing, and that's not the important part, and it also leaves a lot of room for human error, like off by one or you put a comma instead of semicolon where you're supposed to, so with \_.each, it really simplifies that where you only have to pay attention to the data and then the function in which you want to run , in the functionality. So, that in Underscore, each part, where it says function(name), that's referring to each of the objects in the animal names or just a placeholder? Here, yeah that's a placeholder. This is a placeholder that represent each of the iterations of animal names. Exactly, so it's val. And then you'd have to put val down there. Right, exactly. It's gotta be the same. Yeah, and the cool thing about this is that we can say name, right? It's just more clear versus animalNames at i. It's less clear what that means if we just say, oh if we call this name, then it's clear to me, oh so we're making an animal, we're passing it a name, and then we're adding it to the farm. So it's just more clear what you're doing, and you're able to do that name in here versus when we have a regular loop where we have to do all this bracket i stuff. So, the argument to the callback function refers to the umm, index in the list, each index in the list? Yeah, so the parameter here could be, is going to be, the first one's going to be Smokey, and you can also put the index here, and you can also put the whole list, but since we're not really using it, we don't need to do that, but you could. So every time it loops, the name will go Smokey, Fluffy, Trigger. Yes, yep. What if animal names, that array, instead of just having a string in each area or each spot, what if it was an object with, you know what I'm saying? What if it was a more complex object, how would each know what to pick out of there? Yeah, so there's a few ways you could deal with that, so you could say, you could call this, then this would be like animal, call it animalObj, in case you forget it's an object. And then you can actually just nest your \_.each, like this and this would loop through your animalObj, and then in this function you could do whatever you wanted to do with it. So this would be like, (typing) So you could nest your \_.each statement like that. So in that case, when you're passing an object, first argument's going to be the property name, and then the second argument's going to be the value? No, it's going to be the value and then the property name and then animalObj, yeah, so we're, for an array, it would be the index, for an object, it would just be the property name, itself. And again, if you were nesting regular four loops, then you start to have things like, you know, you'd probably have a j in there, and it gets really complicated. Here, if for example, this is like, I can't even think of what it could be like. Typeway? (mumbles) Yeah, you can call it like characteristics, or we'll just call it, char for short, and then you would just, it's easy to see that, we're going to like, now we're console.logging all the characteristics of our animalObj versus something that i, j, you know. Cool, so awesome, so we're pushing our animal. Now what if we wanted to then, so any questions about this? Pretty straightforward... awesome.

\_.map()

So another one that we use a lot is \_.map, and \_.map and \_.each have a lot in common. The important thing to know about \_.each is that you can't return anything from \_.each. So, if we see here, we're pushing to our farm, but we're not using a return statement. You cannot use a return statement in this function. It won't work so just keep that in mind. However, in \_.map, it's really important what you return, so \_.map does the same thing where it's going to, it's going to loop through an object or an array, and it's going to apply it to a function. The difference here is what we return, so if we return here, we return some value here, that's going to be placed into an array, and so we have to save this, in a variable, and so nums would then be again, just an array of numbers, so it would return that. So whatever is returned from the callback is going to be put into an array. So we can return num the value plus one, and then that would output two, three, four. So then you can transform your data this way. So, it produces a new array of values by mapping each value in the list, and transforming it, using this function, and the same holds true with the parameters are always going to be the value, the index, and the entire list, cool. So back to our Charisaur, sorry our pocketmon example. We have an array of pocketmons, and then we have a function called excitedArr, and this is my algorithm that takes any string and suddenly makes it an exciting string, so how do we use this, using .map? So of course we'd say \_.map, and what do we pass it? How about Tanner? Pass it a one, two, three in the brackets, or no, no, no I'm sorry. Yeah, you're one slide behind. Repass it the function pocketmon. Yeah, so we'll pass it pocketmon, which is an array of our pocketmons. So would it be pocketmons or pocketmon, what do you think? Pocketmon. Pocketmon, for plural, so it's like sheep, sheep, okay. Or deer. Deer, yeah, cool. And then what do we pass? What about William, you've been quiet. Got to set that callback function, you want to call for each one of them so I think it's an excitedArr. Yep, there is no other functions on the page, and so we just pass excitedArr, Notice we're not calling it here, we're simply passing it. That's important, cool, shall we test it? How does it know that it's going to use the value, cause when you're defining excitedArr, it's just using val? Yeah, so that's just a parameter. We can call this like string two. I know but umm, maybe I-- It just the, the mechanics of \_.map, how it works, is that the first thing it calls the function with, whatever function it is, whether or not it has parameters or not, It's going to call excitedArr like this. If we're just doing this example, it would be pocketmon, @, you know the first one would be zero, yeah it would be zero, and then zero, and then it would be just pocketmon. So that's what it's doing under the hood, it's calling it like that every time, and then the second time, it calls it like this, then the third and final time, it will call it like that. So and because pocketmon that gets substituted for str? And the rest just doesn't get, it gets discarded? Totally, so the first argument is going to map to the first parameter, and since str is first parameter, str is now pocketmon Question is, is passing the function in the \_.map, the same as calling it? So, is it the same as calling it, so when you pass it to \_.map, you aren't calling it, but \_.map calls it for you, so yes and no. If you don't want this function to be called, then you shouldn't be passing it to \_.map because \_.map will just, under the hood it calls the function for you, really similar to in our callback exercises where we took a function and then we called it immediately. So is it, that one that you have highlighted, that's just a reference, it's telling \_.map, this is where that function is, it's not actually doing the function, but then when \_.map does its internal things, it's going to that address and saying here's the function. Exactly, so \_.map is its own function, so we're calling \_.map and somewhere in the Underscore library, we're entering into the function body of \_.map, right? And we have two arguments that we're giving to \_.map. And \_.map has some functionality that it does always with these two arguments, so inside the function body, it's going to loop through this list and it's going to call this function with each value of the list, so as it loops, it's going to call it and that's how it works under the hood and we can even look at it if we wanted to, \_.map. So, we see it takes an object and it takes an iterator, and if we look somewhere in here, we see that it's, it's calling the iterator function, so it's looping first of all, right, see that loop, and it's calling, it's doing a little bit more than, than what I'm expressing, but the main point is that it's looping, right, and then it's calling that iterator function with the value, with the property, and with the entire object, right? And so that's what \_.map is doing under the hood. Cool, any questions on that? And the key thing here is we want to say this, save this in a variable because it's returning an array, so var excitedPocketmon equals this so now, we have transformed our array into an excitedArr. Could you have saved it in your same Pocketmon, one to overwrite and that should be okay?

Looping with \_.map()

Cool, so knowing what we know now about \_.map, maybe it makes more sense, instead of using \_.each, to rewrite this using \_.map. So, that's our next step and I'll give you about five minutes to get started with translating this, each function into a \_.map function that saves all of our animal objects into our farm. Alright, so what's the first thing that we need to do here? Maybe do a \_.map, what are we going to \_.map over, do you know John? The animal names. Mm-hmm. Oh no, I ran out of animal names. So, we're mapping over the animal names, what function do we want to put here? What is it? Speak No, oh this is speak, no we want to, we definitely want to call AnimalMaker somewhere. \_.map will call it for you. If you're just passing AnimalMaker, \_.map is going to call it with the name. That's true, let's do it out the long way, and then we'll clean it up. So we're passing, what's being passed here? Name of the animal. Mm-hmm. And then what do we want to do next? Call AnimalMaker with that name. Yep, absolutely, name and then what else? There's a couple small minor things that we need to do. Well, we got to put in the farm, which you can do, farm = And then one last but very, very important thing. Return. Yep, we need to return it. Awesome, so now we see we've kind of condensed a line or so, we have our animal names, and we're passing each name, and we're returning it and now we don't have to do this extra operation which is pushing it to the farm, it just happens automatically, which is awesome. And as John mentioned, we can even go a step further, and just not even have this function, and we can just put AnimalMaker directly here. And then it will just call it with the name as the first parameter, so with that we could just leave it like that, and that's even shorter and then we can see, if you understand the mechanics of how \_.map works, you know already that you're using animal names and you're making an animal with each animal name. So, that even makes it more clear to yourself, when you're looking back on the code and you're like, what does this code do again, I forgot, or to your colleagues who are going to pick up your code after you. And \_.map's super common, everyone pretty much uses it, and should understand it, mm-hmm? So, I'm confused by what the return value of AnimalMaker is, I mean it's an object, right, but what, I guess I'm confused how this works. Sure, so whatever this function returns is going to be stored as an array, and then we're going to store that array, it's going to be farm, so \_.map here, is going to return the array of whatever the callback function returns. So our callback function is AnimalMaker, so for the first loop, the first iteration with Sally, we're going to call AnimalMaker, we're going to pass the value of Sally, so it's the name to the AnimalMaker. We enter into this function body, and it's returning an object with one property on it, that's a property called speak, and which is a function that all it does is console.log "my name is ", and it says its name. But we're not calling speak, right? No, you're not calling speak, you're calling AnimalMaker and it's returning-- We're not doing anything with the animals once they get back into the farm array, so that's why I think we're kind of struggling with, the name of the function is AnimalMaker, yet it's really making-- It's making animal objects. Oh, so it's storing the animal objects in the array. And we're giving our animals the ability to speak. So if we never use this function, we'd all have just like really quiet animals, which might be better, I don't know. Now if you did access, let's say, the first index of the farm list, it'll basically spit out in a console, hi, my name is whatever that first, Sally or whatever. Mm-hmm, so we can just go ahead and delete this, and we can run this code and we can kind of look at it. And so it's important wherever you're running this code that you have the Underscore library included because it's a library, if you just run it on a new site, these methods aren't available but luckily we know for sure that the Underscore library site has Underscore on it. That'd be funny if it didn't, but. And so, let's check out our farm. So, we see our farm is three different objects with a speak function on it, and then if we wanted someone in our farm to speak, what we could do is we could say the first one .speak, and it says my name is Sally, and then we could try the next one and it will say their name, like that. And then we can investigate, that's just an object with one property, speak. Is that clear? And for those on chat who are asking about the explicit return, we are explicitly returning something here. And then also, inside of the \_.map function, it's also returning in there, so.

\_.map() vs. \_.each

So, then let's talk about the difference between \_.each and \_.map. So, the important thing to note is that \_.each, like I've mentioned before, you can't return anything from \_.each. So \_.each would be something better suited for getting each animal to speak versus if you wanted to have it return something, especially if you wanted to return an array, \_.map is a better bet, however if you don't want it to explicitly return an array, maybe you want it to return an object, you might want to wrap \_.each into a function where it pushes to an object instead. So, then we can run all of our code or let's get some more names, let's see. So if I wanted to \_.map that into, if I wanted to use \_.each to \_.map it into an object, I still have to make a function, but I don't have to iterate cause \_.each does that for me, and just push it onto that object. Yeah exactly, however you want to implement it, but \_.map is always going to return an array. Question, the prior example had an explicit return on AnimalMaker, why the difference? What do they mean by explicit return? When you use \_.map, you have to have return, but when you use \_.each, you don't use return. Yeah, when you use \_.each, you can't use return. \_.each doesn't return anything, but \_.map you have to use return. Whatever is returned from that callback function is then going to be put into the array. So, it's really important that whenever you're using \_.map that you're actually returning something otherwise you're going to get an array of undefined, so if I didn't return here, we'd get, and we still mapped over it, we'd get an array of three undefined values. So, we need to make sure that we return in that callback function, whereas an animal in the, for the animal.speak is fine because we're not returning anything there, we're just console.logging. They're a little bit behind us, but the prior \_.map example didn't have a return. Or maybe did I miss the return? I think when we did that one, I think what they're saying is when you did the \_.map with var farm there was no return there, is that, maybe that's what they're. Like this, there needs to be a return, so if there wasn't a return, I made a mistake. Here, AnimalMaker, name, so you have to return here, because what this is saying is, so AnimalMaker returns something, and then you're returning whatever AnimalMaker returns. So, that's why there's two returns here. So I hope that answers their question. Okay, cool. We're right on time again, awesome. So, just a quick recap, \_.each we can't return anything, \_.map we have to return something, and it returns it to an array, did you have another question Mark? Yeah, they're just saying that by just putting AnimalMaker there that implicitly has the return? It explicitly has a return inside AnimalMaker. Right, but on that \_.map, I think the last example, the short version just had, yeah AnimalMaker, so that implicitly doesn't return, right? Yes. Okay. But it's important that you actually explicitly return something inside AnimalMaker, that's the difference. Since AnimalMaker's already returning something, you didn't need that intermediate anonymous function to return the value of calling AnimalMaker because it already did that. Yeah, exactly.

Underscore Exercise

And then the exercises that we're going to do is in here. We're going to do Underscore loops here. And there's going to be things in here that aren't, that we didn't cover like \_.filter and you can just read the Underscore documentation to find more information on how \_.filter works. And if you finish ahead of time, I have more Underscore exercises that you can dive into.

Underscore Exercise Solution

So we're going to use an \_.each loop to loop through an array and console.log the different values, and the way that we would do that is here we have our array, and here's our callback function here. We wanted to console.log the value, so this is going to be the value. This will be the index, and this one would be the list. So if we want to console.log the value, we do that. And then if we wanted to console.log the index, we would do it that way, and how would this be different if you were looping through an object? Anyone? Do the properties? Yeah, so the main difference, there's not really a big difference. The only difference is that instead of the indexes, it would just be the properties, but you could still call it index and it wouldn't matter, cool. So, our second function is called checkValue. It searches an array for a certain value. If it contains that value, it will return true. Otherwise, it will return false. So here we are with just our regular four loop. We are initializing it i at zero, and we're going to loop while i < arr.length, and then we're going to plus one every time. And for every index of the array, we're going to run this block, so we're going to say if array at i, so the first one, is zero. Oh, I should do this, so checkvalue. So we're going to check the value for bye. So, it's going to say, if array at i, the first time is going to be zero. So if bonjour is the same as bye, oops I'm sorry, we have to do this. Let's start over, so we're passing helloArr. And we're also passing bye as the string that we're checking to see if that is in the array. So we're looping through, so the first one would be bonjour, and the value that we're checking against is bye. So, that doesn't equal each other, so we won't enter into this block here. We're going to skip this, and then we're going to run it again, and it says, if hello = bye, and it doesn't so it's going to skip it again, and we'll run this block one more time. And it says, hola, so it does, hola equal goodbye. No, so then it's going to skip this if block, and then this four loop is done. So then we're going to return false. However, for hello, let's just do bonjour, for bonjour, we are going to say, we are going to loop through the array, and if bonjour equals bonjour, which it does, we're going to skip in here, and we're going to return true. The important thing about returning true, right here, is that it's going to break out of if, and it's going to break out of the four, and it's going to break out of the function. So, the entire function at that point will return true, it's going to stop looping, so be really careful about where you're returning, if you're returning inside a four loop, you want to make sure that you really want to stop looping at that point where you have that return statement. Cool, and then so that same thing with bonjour. Going to put this down here, I'm sorry, with wow, I'm sorry. With each, it's been a long two days of teaching. So, the same checkvalue with \_.each, and how we're going to do that is because we can't return anything from \_.each, we have to initialize the value before it and return it after. So, we're going to loop through, and we're going to check if the value, the first value equals val, etcetera, etcetera. If it does, we're going to say result is true. But this won't stop the loop, it will keep looping. And then once we're done looping out of the \_.each, we're going to return the result, and so result is false unless proven true, and it will be proven true in here, otherwise it will just remain false, and that's fine. We're going to return it down there. Any questions about those first few ones? Looking at Underscore, it also has a sum function. Underscore has a sum function, like s-o-m-e. S-o-m-e, which will break off the first time it returns false, er true? Cool, yep. More efficient than \_.each. So Underscore has tons and tons of awesome utility functions that you can go through and once you get more familiar with the library, these things will just come to you. Cool, so we're going to write a loop that pushes all the values and object to an array. We're going to skip this and we're just going to do the \_.map, and so we're going to say var, what is it, myNums equals \_.map, and so a \_.map is going to loop through this object, and then the first parameter's the value, the second one is going to be the property name, right? So just for reference, and then we're just going to return val, so val is going to be two, four, three, and this is the number, not the string, and 12. However, if we wanted to return the property names, we could just return prop, and that would return an array of property names. Cool, any questions on that one? Cool, so then the last one is \_.filter. It's really similar to \_.map and \_.each, except \_.filter only returns, only returns the true, hold on let me think how to explain this. So for the callback function, \_.filter just returns, it will just add it if it's true. Okay, actually let me double-check that, now I'm starting to doubt myself. Let's look up \_.filter, so it loops through the list. Oh yes, so pass a truth test, so if, oh I see, so I actually implemented it wrong, excuse me. So we can just return, it will give the value if this is true, so val mod two will equal zero if val is even. And then we can double-check this. Just to make sure, uh-oh syntax error. The curly bracket at the end, I think. Thank you. Of course, so I need to, whenever you see this error, \_ is not defined, that means you don't have the library in that window, and you can always do a quick check here. Whoops, not that one, just by, you can literally just type \_ and for some reason on that annotated source code page, we don't have Underscore loaded, but if we check here, we see that we have Underscore. It just returns an object or a function, I'm sorry. Okay, val is not defined, val, there we go. So that's \_.filter, so \_.filter is going to push whatever values in the array, or I'm sorry, in the object or the array, into an array if this function returns true. So, val mod two is true, is equals equals zero when it's even, does everyone know how the modulo works? Thumbs up, modulo, okay cool, awesome. So do you have any questions on those exercises? Let me put the correct answer. Yeah, I have a question on line 43 there. So .\_map runs through your object, pulls out each property that you're returning, and then that looks like magic to me, that it puts it into that array. It looks like magic, is that a question? (laughing) Is it magic? Where's myNums? myNums is declared right there on line 43, right? Right. So we're not telling it, it's an array, it's an object or anything, so JavaScript-- Well, \_.map always returns an array, so that's just-- \_.map does? \_.map, yeah the mechanics of \_.map is that it's always going to return an array, and that array is always going to contain the values that are returned from this function. So since we're returning the val here, we know that that's going to be an array of the values, and \_.filter's really similar except that if the value returns true, it will save that into an array. So it's still looping, the only difference with \_.map is that it's the array's going to be the length of the original array or object, so this length is important. So we have four, four properties, so we know that for \_.map, it's going to return an array with four indices, and \_.filter is cool because you can sort of decide on how long that's going to be. So you couldn't use \_.map to \_.filter, for example, even though it kind of seems like it cause \_.filter also returns an array, and \_.map returns an array. \_.map, it always, just the mechanics of the loop, it's always just going to return the same length, and since \_.filter is shorter usually, or the same size or shorter, you can't use \_.map. So will you get a bunch of like undefined gaps, so like if you did the modulus on three, that would be undefined, stuff like that? Yeah, yeah, so then the difference here is that this is actually just returning true or false. This isn't returning the value, you know. Where here, you actually have to explicit return the value. If we did this in \_.map, it would just return, it would then be an array of true or false values. Just went away, awesome, so. That concludes our everything, yeah, thank you guys so much for coming. I'll stick around for more questions. Just a couple things, Mark will be sending out a link to submit for solutions, I'll submit them all in bulk later today, what else, so we teach these classes live at Hack Reactor about once a month, we have a different class that we teach to the public. You can find that on Meetup, so www.meetup.com/hackreactor, we have all our events there. We have regular meet-ups too, study groups, etc., and we also teach these classes through there, all live, in person, in San Francisco. Yeah, thank you guys so much, and I really appreciate everything.