October 20, 2021

Week 2

Focus on what the code is doing, not the synthax.

**Functions 2**

**Intro**

**Overview**

* Returning Values & Syntaxes(we are going to learn a new synthax on how to write a function)
* Arrow Functions(a lot cleaner and modern, etb 2006, important concept)
* Callbacks(call back functions) when we pass a function in another function(parameters in parentheses) the function that is passed inside a function
* Higher Order Functions(function that receives the call back)
* Higher Order Array Methods

**Objectives**

* Student can pass a function as an argument into another function.
* Student can invoke a parameter whose value is a function.
* Student can write a higher order function.
* Student can create/define functions using arrow function syntax.
* Student can use the following array methods: forEach, map, filter, and reduce.

**Returning Values & Syntaxes**

**Returning Values**

* functions that have a return keyword output the value specified in the return statement
* we can store that value in a variable to use elsewhere in our code
* we’ll use this feature later in this lecture

**function** add(x, y) {

**return** x + y<<going to give back whatever is being return in a variable somewhere else on the code>>

}

**function** subtract(x, y) {

**return** x - y

}

**const** numA = add(54, 627) {add is the function…now numA is 681 now}

must have return in order to add those numbers. Console log won’t add them inside the function.

**const** numB = add(235, 8654)

**const** numC = subtract(numB, numA)

console.log(numC) *// 8208*

**Function Syntaxes**

In JavaScript, there are 3 ways to write functions:

* function declaration[how we wrote function so far]
* function expression
* arrow function

**Why?**

* they load and behave a little differently from each other
* the details are more of an intermediate topic

**Declaration**

* this is the traditional way to write functions

**function** giveMeFive() {

**return** 5

}

**Expression**

* function expressions are saved to variables

**Remembering the Difference Between Declaration and Expression**

Think of how a function expression has an equals sign. Expression and equal both start with “E”.

**const** sayHi = **function**() {

**return** 'Hi!'

}

**Arrow-an abbreviated function expression**

* arrow functions are a shorter way of writing function expressions
* they are especially convenient for writing functions in-line

**const** sayBye = () => {

**return** 'Bye!'

}

* this might not look shorter, but fear not, let’s keep going

**Arrow Functions**

**Parameters**

* arrow functions accept parameters just like any other function
* parameters must be in parentheses unless there is **exactly one**
* you can still include them even if there is only one parameter

**const** returnParam = item => {

**return** item

}

**const** makeArr = (one, two, three) => {

**let** arr = [one, two, three]

**return** arr

}

**Implicit Return**

* if you have a function that doesn’t require many steps, you can write one-line arrow functions
* we don’t have to explicitly use the return keyword in one-line arrow functions

**const** giveMeFive = () => 5

**const** addFive = num => num + 5

**More Details on Implicit Returns**

If you are returning an object in a one-line function, you need to wrap the object’s curly braces in parentheses so that they’re not mistaken for the curly braces that hold function blocks.

**const** makePriceObj = number => ({price: number})

You can also wrap implicit returns in parentheses and still have them on a new line. This might seem like a weird strategy but could come in handy when you’re returning something like an object or even another function (foreshadowing the higher order functions) or some HTML!

**const** makeLargeObj = (str, num, arr) => (

{

word: str,

integer: num,

list: arr

}

)

**Callbacks**

**What is a callback?**

* a callback is a function that is passed to another function and invoked within the outer function
* being a callback doesn’t change anything about a function’s definition, it’s just a different way of using the function
* like many subjects in more intermediate and advanced programming, “callback” is a specific name for a certain pattern

**Why use callbacks?**

* they make the outer functions more flexible!
* functions in general help code to be more reusable
* they are required by other functions

**How do we use callbacks?**

* one way is to define the callback outside of the function where it will be invoked

**const** innerFn = () => 'I am a callback!'

**const** anotherInner = () => 'Hello there'

**const** outerFn = callback => console.log(callback())

outerFn(innerFn)

outerFn(anotherInner)

* another way is to write the callback function in line when you invoke the outer function

**const** checkNumValue = (num, cb) => {

**if** (num > 10) {

cb(**true**)

} **else** {

cb(**false**)

}

}

checkNumValue(12, (bool) => {

**if** (bool) {

console.log('you sent in a big number!')

} **else** {

console.log('you sent in a small number!')

}

})

checkNumValue(3, (bool) => {

**if** (bool) {

console.log('your number is bigger than 10')

} **else** {

console.log('your number is smaller than 10')

}

})

**Flexibility**

In both of the examples above, we were able to call an outer function multiple times passing in different callbacks! Callbacks allow us to separate out some functionality so that we don’t have to repeat it. And means that that outer functions can do different things by calling different callbacks. Pretty fancy!

**Higher Order Functions**

**What are higher order functions?**

* a function that takes 1+ function(s) as an argument and/or returns a function
* they have similar benefits to callbacks: flexibility and reusability
* you’ve already seen them: they’re the outer functions in the callback examples

**Receiving Functions as Arguments**

* this the more common form, and one we’ll use often throughout foundations
* all of the outer functions in the callbacks examples were this type of higher order function

**Returning Functions**

* functions that return other functions are like function factories, similar to how classes create objects
* the returned (inner) function has access to the outer function’s parameters

**const** createAdder = x => (y) => x + y

*// here's the same function written as a declaration*

**function** createAdderDec(x) {

**return** **function**(y) {

**return** x + y

}

}

**const** addFive = createAdder(5)

**const** addTen = createAdder(10)

addFive(12) *// output: 17*

addFive(40) *// output: 45*

addTen(3) *// output: 13*

**Higher Order Array Methods**

* functions built into JavaScript that accept callbacks as arguments
* we will cover forEach, map, filter, and reduce
* note: not all higher order functions have to work with arrays, there are others (setTimeout, addEventListener, etc.)

**How They Work**

* each of the methods we’ll cover accept a callback
* all of the methods loop over the given array and perform actions according to the callback function we create

**Callback Parameters**

* forEach, map, and filter invoke the callbacks passed to them with **up to** 3 arguments: element, index, and array (reduce is a little different)
* element stands for each individual item in the array (could be a string, an object, a number, whatever is in the array)
* index is the index of the current element
* array is the array that the method was called on
* element is the only required one
* since we write the callbacks, we actually get to choose the names of the parameters, so they won’t always be element, index, and array
* it’s often helpful to name ‘element’ after whatever it is that you’re looping over (ex. if the array is storing colors then you could use ‘color’ as the parameter name in your callback)

**.forEach**

* loops through an array and “for each” element, executes any specified functionality
* does not necessarily modify the original array, but can if the callback is specified to do so
* very similar to a for loop, except we have easier access to the values and their indexes because of the callback’s parameters

**let** numArray = [0, 1, 2, 3, 4]

*//without arrow function*

numArray.forEach(**function**(element, index){

console.log(`*${*element*}* is at index *${*index*}*`)

})

*//with arrow function*

numArray.forEach((elem, i) => console.log(`*${*elem*}* is at index *${*i*}*`))

**.map**

* map does not modify the original array
* it returns a copy of the original array, with any changes implemented in the callback applied

**const** numArray = [0, 1, 2, 3, 4]

*//without arrow function*

**const** mappedArr = numArray.map(**function**(num){

**return** num += 2

})

*//with arrow function*

**const** mappedArr = numArray.map(num => num += 2)

console.log(mappedArr) *// [2, 3, 4, 5, 6]*

**.filter**

* also creates a new array, but with some values filtered out
* the callback function needs to return true or false, and will most often return a conditional statement
* if that statement evaluates to **true** the element will be included in the new array
* filter is not destructive (the original array will stay the same)

**const** numArray = [0, 1, 2, 3, 4]

*//without arrow function*

**const** filteredArr = numArray.filter(**function**(element){

**return** element % 2 === 0

})

*//with arrow function*

**const** filteredArr = numArray.filter(element => element % 2 === 0)

console.log(filteredArr) *// [0, 2, 4]*

**Evaluating Conditions**

How are these conditional statements working?

The .filter method is expecting the callback to return a boolean. When we write something like const filteredArr = numArray.filter(element => element % 2 === 0), the element % 2 === 0 portion in the return of the callback evaluates to true or false like any other conditional statement. We don’t have to wrap it in parentheses and put an ***if*** in front of it for it to be evaluated. Since there is a comparison operator in the statement (===), JavaScript automatically evaluates it.

Try console logging a conditional statement like this: console.log(4 === 5) and see what happens.

**.reduce**

* executes a “reducing” callback function and returns a single value
* the callback will be invoked with at least two arguments: the accumulator and the current value (the index and array can also be included as the 3rd and 4th args)
* the accumulator is the value that will ultimately be returned
* we can set the initial value of the accumulator, if we don’t it will default to the first value in the array
* reduce is often used to create a total sum, but can be used for other (more advanced) purposes

**const** numArray = [0, 1, 2, 3, 4]

*//without arrow function*

**const** reduced = numArray.reduce(**function**(acc, curr){

**return** acc + curr

})

*//with arrow function*

**const** reduced = numArray.reduce((acc, curr) => acc + curr)

*//with initial value*

**const** newReduced = numArray.reduce((a, c) => {a + c}, 10)

console.log(reduced) *// 10*

console.log(newReduced) *// 20*

**Chaining Methods**

* methods can be chained together, allowing you to perform multiple operations on one array concisely

**const** numArray = [0, 1, 2, 3, 4]

**const** myNum = numArray.filter(num => num > 2).map(num => num \* 3).reduce((acc, curr) => acc - curr)

**The End**

© 2021 Devmountain