Javascript I

Function Declaration

Function declarations are made up of several syntactical parts:

1) function keyword --> function

2) name of function

3) list of parameters(optional)

4) statements inside the block of code --> {}

Functions need to be called for the execution to occur.

This is called invoking a function

Example:

// Define our function declaration

function add(a,b) {

return a + b; --> this is called a return statement

} anything written after a return statement will not execute

it terminates the function after running the line of statements within it

// Invoke the function add

console.log(add(2,4)) // 6 --> console.log() is used to see the result in our web console

when we invoke a function we pass in arguments

they are received into the parameters of the function in the order they were called

\*\*\* arguments are values that are passed into functions that receive them as parameters \*\*\*

Function Declaration Hoisting

JavaScript uses a 2-pass compiler when executing line of code.

When we run JavaScript in the browser, 2-passes will take over our code.

1) Sets up references to all our code

function declarations are defined in the first pass

2) Applies the values to the references that were found

compiler is made aware of the function declaration

and adds it to the top of the execution order for your code

Hoisting -putting a function higher in the execution order for later use

\*\*\* This means function declarations can be invoked BEFORE they are defined \*\*\*

Function Expressions

Unique differences when compared to function declarations:

1) A variable is used to store the function for later use

2) Anonymous functions are used

3) Not hoisted

can only be invoked after a definition has been placed in the execution stack

Example:

const add = function(a,b){ --> No need for a name because we can use the name of the const 'add' as a reference to the invocation

return a + b;

}

console.log(add(2,4)) // 6

Why use this?

1) We have more control over our code

2) We can tell it when to do its job easier

3) syntactical sugar

4) We can place functions inside functions easier without having to name them

Arrow Function Expressions

1) Function Expressions with the this keyword removed

Example:

// function expression syntax

// const add = function(a,b) {

// return a + b;

// }

// arrow function expression syntax

const add = (a,b) => {

return a + b;

}

console.log(add(2,4)) // 6

// streamlined arrow syntax

// Because we only have one return statement, no need for return keyword or {}. They are implied.

const add = (a,b) => a + b;

console.log(add(2,4)) // 6

When NOT TO USE Arrow Functions:

1) Event Handlers(Unless they are inside a class constructor)

2) Object Methods

3) Prototype Methods

4) Anytime you need to use arguments Object

---------------------------

let, const, var

Example:

var name = 'Batman';

What have we done here: This is how we capture data into memory in JavaScript

1) told the computer to hold a String: ‘Batman’ in memory

2) to retrieve that information call the ‘name’ variable console.log(name);

we should get the string ‘Batman’ printed out in the console

Because JavaScript is a dynamic language. We actually can override what the variable name points to in memory.

Example:

var name = 'Batman';

var name = 'Robin';

console.log(name); --> prints 'Robin';

-----------------------

Example:

let name = 'Batman';

name = 'Robin';

console.log(name) --> 'Robin';

Learnings:

1) let is --mutable--

meaning we can change the value (or thing that a variable points to) that a variable declared with let points to

2) the name variable has been --re-assigned-- but not --overridden-- by another name variable declared with the let keyword

name now points to the String:’Robin’

-----------------------

const is the most strict of all keywords used to declare variables in JavaScript

const is a keywords that we use to declare a variable

you can never re-assign what a constant variable points to like you can with let and var

THIS EXAMPLE IS NOT POSSIBLE:

const name = 'Batman';

name = 'Robin';

console.log(name) --> 'Robin';

IT WILL THROW AN ERROR

-------------------------

Recap:

1) var can be re-assigned and overridden

2) let can be re-assigned but NOT OVERRIDDEN

3) const CANNOT be re-assigned nor overridden

\*\*\* When defining/declaring variables use const until you can’t, then use let \*\*\*

-------------------------

Object Literals

\*\*\* Everything in JavaScript is an object \*\*\*

Object Properties

Objects in JavaScript are used as a way to store data and give the programmer access to that data when needed.

This ability to store data and call it is known as a property.

Properties are organized in a key:value pairing.

Example: Object Literal with some properties on it

const myPersonalObject = {

firstName: 'Fred',

lastName: 'Flintstone',

};

1) keys of myPersonObject are firstName and lastName

2) values of those keys are Fred and Flintstone Respectively

If we want to use myPersonalObject later on, all we have to do is reference it with either

bracket notation

dot notation

Bracket Notation for referencing the firstName property is:

myPersonalObject["firstName"] // 'Fred'

\*\*\* Pay attention to the quotes around the “firstName” property.

This is because we’re using brackets around the firstName value. \*\*\*

Dot Notation for referencing the firstName property is:

myPersonalObject.firstName // 'Fred'

Methods for Handling Objects:

Object.keys() -> Gives us an array back of the Object's properties/keys

const keys = Object.keys(myPersonalObject);

returns ['firstName', 'lastName'];

Object.values() -> Gives us an array back of all the Object's values

const values = Object.values(myPersonalObject);

returns ['Wilma', 'Flintstone'];

Object.entries() -> Gives us back an array of the Object's key/value pairs as a `tuple`

const entries = Object.entries(myPersonalObject);

returns [ [ 'firstName', 'Wilma' ], [ 'lastName', 'Flintstone' ] ]?????

---------------------------

Arrays

Arrays have a special 0 based indexing (organizational) ability. They allow us to store data in them sequential based on 0.

Access any item in an array by selecting the array, and using ‘brackets’ to select which index you want to reference.

Example:

const hogwarts = ['Harry', 'Hermione', 'Ron'];

hogwarts[0] === 'Harry';

hogwarts[1] === 'Hermionie';

hogwarts[3] === 'Ron';

add a character to our hogwarts array (adds to the end of the array):

.push()

Example:

hogwarts.push('Dumbledore');

// gives us -> ['Harry', 'Hermione', 'Ron', 'Dumbledore]';

add something to the front of the array (adds to the beginning of the array):

.unshift()

Example:

hogwarts.unshift('Snape');

// gives us -> ['Snape', 'Harry', 'Hermione', 'Ron', 'Dumbledore]';

to remove the last item in the array:

.pop()

Example:

hogwarts.pop(); -> this will give you the item poped off the array.

// gives us -> ['Snape', 'Harry', 'Hermione', 'Ron']';

to change the contents of an array by removing or replacing existing elements and/or adding new elements in place

.splice()

Examples:

var months = ['Jan', 'March', 'April', 'June'];

months.splice(1, 0, 'Feb');

// inserts at index 1

console.log(months);

// expected output: Array ['Jan', 'Feb', 'March', 'April', 'June']

months.splice(4, 1, 'May');

// replaces 1 element at index 4

console.log(months);

// expected output: Array ['Jan', 'Feb', 'March', 'April', 'May']

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

JavaScript II

Closure needs to be discussed through the lens of this statement “What data do I currently have access to in my program”.

Global vs Local .Scope

When a function is declared and created, a new scope is created.

Any variables declared within that function’s scope will be enclosed in a lexical/private scope that belongs to that function.

functions look outward for context.

Closure

\*\*\*if a variable cannot be found in the lexical scope of a function, the function will look in the outer scope\*\*\*

Example:

const foo = 'bar';

function returnFoo () {

return foo;

}

returnFoo();

// -> reaches outside its scope to find foo because it doesn't exist inside of return Foo's scope when foo is referenced.

-----------------------------

Callback Functions

Callback functions are just functions that are passed into other functions as arguments.

Functions in JavaScript are a Type like any other Type (Boolean, String, Number)

and they can be passed around as arguments to other functions.

Examples:

const functionFeeder = function(callback) {

callback('Hello from the inside of Function Feeder');

};

What we have done here is define a function called functionFeeder and passed it a parameter called callback.

So when functionFeeder is invoked, we have the ability to feed it a function as an argument.

That function in turn, will feed back the string Hello from the inside of Function Feeder to the callback function it receives.

And thus, when we call functionFeeder we can handle that string that comes back to us like so:

functionFeeder((string) => { // invoking the function

alert(string); // alert a function that pops up a box in the browser.

});

functionFeeder respectively becomes what is known as a higher order function or

a callBack function allowing it to take in a function as a parameter

and executes a function when called with a function as it’s argument.

Example:

function sayHello(name) {

console.log( ‘Hello, ${name}’);

}

sayHello is a function that says hello to a name that is passed into it.

Now let’s set up a function that will define the name passed into sayHello and pass it into our function.

function sayHelloWithLars(callback) {

const innerName = ‘Lars’;

callback(innerName);

}

The callback will become sayHello when we call it like this:

callSayHelloWithLars(sayHello);

callSayHelloWithLars executes sayHello as its callback.

Built in JavaScript function methods

.forEach(), .map(), .reduce(), .filter()

.forEach() is an iterator function in which we use a callback to access the items in an array that come back with each pass.

Example:

const items = [‘feather’, ‘coupon’, ‘cup’ , ‘drill’];

for (let i = 0; i < items.length; i++) {

alert(items[i]);

}

Or

items.forEach(item => alert(item) ); // item => alert(item) is the callback

const elements = ['earth', 'wind', 'fire', 'water'];

Create a function called show first that passes back the first item in the given array

function showFirst(array, callback) {

callback(array[0]);

};

showFirst(elements, (firstItem) => {

alert(firstItem);

});

Create a function like ‘shoFirst’ but this time, show the length of the array passed in:

function shoLength(array, callback) {

callback(array.length);

};

shoLength(elements, (shoLength) => {

alert(length);

});

Use forEach again to loop over our array and alert each item in the array:

elements.forEach(element => alert(element))

.map, .filter, .reduce

const data = [

{"city":"seattle", "state":"WA", "population":652405, "land\_area":83.9},

{"city":"new york", "state":"NY", "population":8405837, "land\_area":302.6},

{"city":"boston", "state":"MA", "population":645966, "land\_area":48.3},

{"city":"kansas city", "state":"MO", "population":467007, "land\_area":315}

];

Task 1: Map the data to a new array, with objects of the city and state names.

const cityStates = [];

for(let i = 0; i < data.length; i++) {

let mappedObj = {};

mappedObj.city = data[i].city;

mappedObj.state = data[i].state;

cityStates.push(mappedObj);

mappedObj = {};

}

const mappedCityStates = data.map((state) => {

return {'city': state.city, 'state': state.state};

});

The biggest difference between .forEach and .map is that map returns a new array of elements while in turn passing each element back to the callback.

Remember that our map function takes in a callback that passes back a couple of things to us. The three things you’d get back from a callback passed to Map would be:

The current item of the array state

The current index of the current item index

The entire array data

const mappedCityStates = data.map((state, index, data) => {

return {'city': state.city, 'state': state.state};

});

.filter

filter returns an item that passes what is called a truth test.

What if we want to only see states who’s population is greater than 650000?

const largeStates = [];

for(let i = 0; i < data.length; i++) {

if(data[i].population >= 650000) {

largeStates.push(data[i]);

}

}

const filterLargeStates = data.filter((state) => {

return state.population >= 650000;

});

On this line return state.population >= 650000; we are explicitly returning an object who’s population is greater than 650000, and in turn passing that object to a new array.

Think of that line reading like this: IF state.population is greater than or equal to 650000 then push it into a new array else do nothing.

.reduce

if we want our data reduced to a single value, we can aggregate that data and use .reduce to do so for us.

let statePopulations = 0;

for(let i = 0; i < data.length; i++) {

statePopulations += data[i].population;

}

const reduceStatePopulations = data.reduce((total, state) => {

return total += state.population;

}, 0);

First of all, we’re passing 0 as a second argument to our reduce function. This argument will become the starting value of our total and if not provided would default to the first item in the array. In this case that would be disastrous because the first item of our array is an object, and we’re trying to reduce our total to a single numerical value. Thus, providing a starting value for total sets what our data type will reduce into. Remember that total also gets memoized or remembered by our function each pass.

The four items that get passed back from our callback function when using .reduce are:

* The current value of the total aggregated value.
  + We set the initial value at the end of the function. In this case we set it to 0.
  + This could be any value though.
* The current item in the array.
* The index.
* The full array.

The this keyword

* You can think of this, as a Pointer to an object. For example: you can use the this keyword to reference an object without having to refer to that object’s name.

Principle 1: **Window/Global Object Binding**

* When in the global scope, the value of “this” will be the window/console Object.

function sayName(name) {

console.log(this);

return name;

}

sayName("D'Artagnan");

Principle 2: **Implicit Binding**

* Whenever a function is called by a preceding dot, the object before that dot is this.

const myObj = {

greeting: 'Hello',

sayHello: function(name) {

console.log(`${this.greeting} my name is ${name}`);

console.log(this);

}

};

myObj.sayHello('Ryan');

const sayNameFunc = obj => {

obj.sayName = function() {

console.log(`Hello my name is ${this.name}`);

console.log(this);

};

};

const me = { name: 'Ryan' };

const you = { name: 'Freddy' };

sayNameFunc(me);

sayNameFunc(you);

// Invoke Methods on our objects

me.sayName();

you.sayNa

me();

We have a function that receives an object as an argument. Depending on the object being passed in, we get a different context for this so when we log out the this keyword we get a different object each time it’s run.

Principle 3: **New binding**

* Whenever a constructor function is used, this refers to the specific instance of the object that is created and returned by the constructor function.

A constructor function is a function that returns and object. It is an object creator.

function CordialPerson(greeter) {

this.greeting = 'Hello ';

this.greeter = greeter;

this.speak = function() {

console.log(this.greeting + this.greeter);

console.log(this);

};

}

const jerry = new CordialPerson('Newman');

const newman = new CordialPerson('Jerry');

jerry.speak();

newman.speak();

Principle 4: **Explicit binding**

* Whenever JavaScript’s call or apply method is used, this is explicitly defined.

Taking the above object oriented approach we can discover that we can override what the CordialPerson constructor objects get set to. By calling them explicitly with a new context using .call and .apply

jerry.speak.call(newman); newman.speak.apply(jerry);

“A constructor function, ‘Constructs’ objects”. It can be thought of as a template. The function itself needs to take in an object literal of some sort so that it can map that object literal’s properties to a new object that will be returned once instantiated.

function Person(attributes) {

this.age = attributes.age;

this.name = attributes.name;

this.homeTown = attributes.homeTown;

this.speak = function () {

return `Hello, my name is ${this.name}`;

};

}

We call Person with the new keyword, and we feed it an object literal that will map to those attributes specified in the Person block.

* When new is called, the constructor function can essentially create a context for a this object. Then what gets returned from that constructor function is that particular this object with the new properties added to it.
* const fred = new Person({
* age: 35,
* name: 'Fred',
* homeTown: 'Bedrock'
* });

console.log(fred); console.log(fred.speak());

Classes

**Classes in JavaScript are just special functions**.

Class Declarations

class Rectangle {

constructor(height, width) {

this.height = height;

this.width = width;

}

} \*

You can think of the constructor function as the foundation of every class. When present, you’ll be building your Object’s properties (that get returned when new is called) inside the constructor’s body on the this keyword.

Any attributes you’d like bound to your created object as properties can be passed in through the constructor method. In this case we have height & width as attributes that we’d like to store on our created object. Remember:**classes will return us objects**.

if I wanted to use this class to create me an object, declare some variable and assign it the class by using the new keyword.

const newRect = new Rectangle(400, 800);

console.log(newRect);

Logs out:

Rectangle { height: 400, width: 800 }

Inheritance w/ Classes

Inheritance is where classes really shine. The extends keyword, and super();function make it so trivial to bind our classes together to achieve some simple object inheritance. The extends keyword will abstract away any of the Class.call syntax that we’re used to. super() is used to tell a parent’s constructor to be concerned with the child’s attributes vis versa and abstracts away the Object.create(this, Class) syntax that is really tricky to sink one’s teeth into.

class Animal {

constructor(name) {

this.name = name;

}

speak() {

console.log(this.name + ' makes a noise.');

}

}

Notice that speak() is a method with some special sytnax on this class. This method will actually not be a method on the object, but it will live on the object’s prototype instead. Which is nifty when you’re worried about memory etc.

If I wanted to create a sub-class from this Animal class it’s really simple.

class Dog extends Animal {

constructor(name) {

super(name);

}

speak() {

console.log(this.name + ' barks.');

}

}

And using this child class:

const doggy = new Dog('Grizzly');

doggy.speak();

---> 'Grizzly barks.'

Some key notes on teaching Classes

\*The object that gets returned from our class will have it’s attributes assigned to it, and returned out of the constructor() function.

* All methods attached to the class body will be stored on the Objects prototype in a special way. There is a bit more magic here than just Object.create(Foo.prototype); and Class.call(this, attrs); But now that we know this, we can accept that the class keyword does this gloriously for us.
* The ‘extends’ keyword is used to extend a parent object. A clue to find out if a class is a sub-class is to look for extends.
* Finally, IF you’re going to use extends,super() needs to be called from within the constructor function. This is to pass any new attributes back up to the constructor of the parent object.

function Person(attributes) {

this.age = attributes.age;

this.name = attributes.name;

this.homeTown = attributes.homeTown;

}

Person.prototype.speak = function () {

return `Hello, my name is ${this.name}`;

};

class Person {

constructor(attributes) {

this.age = attributes.age;

this.name = attributes.name;

this.homeTown = attributes.homeTown;

}

speak() {

return `Hello, my name is ${this.name}`;

}

}

Concepts to go over.

* Speak is now assigned to the object’s prototype. (Pop open the console and show this off after you instanciate an object.)
* This is a single class, meaning it is not extending a parent class.
* Instead of having to reference the objects prototype over and over to create methods on it, you can simply add them to the class body. This is how we use classes today

Now, where this comes in handy is when we have children objects that will be sub-classes of their parents.

function Child(childAttrs) {

Person.call(this, childAttrs); // this is the special sauce

this.isChild = childAttrs.isChild; // this will be a special attribute to Child

}

Child.prototype.checkIfChild = function() {

if(this.isChild) {

console.log(`${this.speak} and I am a child object`);

}

};

This function above can now be represented as this:

class Child extends Parent {

constructor(childAttrs) {

super(childAttrs);

this.isChild = childAttrs.isChild;

}

checkIfChild() {

if(this.isChild) {

console.log(`${this.speak} and I am a child object`);

}

}

Now make your fred and pebble’s objects

const fred = new Person({

age: 35,

name: 'Fred',

homeTown: 'Bedrock'

});

const pebbles = new Child({

age: 3,

name: 'Pebbles',

homeTown: 'Bedrock',

});