**Week1**

**1.2 JS Foundation**

* JS is an interpreted and dynamic language.
* C++ is a strict language. An example of strict is that it does not allow int variables to be assigned string.
* This is why TS came into the picture to allow JS to be more static.
* JS has single threaded nature which means that if the program has 10 lines of code, the 10 lines would run on a single core and not divide the lines between multiple cores to reduce time. Bad language for scalable systems. We can make it run parallel using cluster module.
* There are 3 ways to declare a variable in JS (let/var/const). Var is archive. We would use let and const.
* Objects in JS are like dictionaries in Python. You can do object.key as well as object[“key”].
* Callbacks - Function can take other function signature as input. The variable type is function. The function’s name without round brackets () is passed as an argument. Key difference is that the signature of the function is passed, the function is not called. See example in code.
* Callback functions must have same signatures. If the functions, you want to callback have different signatures then it is not a use case for callback. It is still possible but defeats the whole purpose of callback.

**1.3 JS APIs**

* str.length 🡪 this is not a function.
* str.indexOf(target)
* str.lastIndexOf(target)
* str.slice(start, end)
* str.substr(start, length)
* str.replace(target, replacement)
* str.split(“”) 🡪 every element is split
* str.split() 🡪 nothing happens, str is returned in an array
* str.split(“ “) 🡪 necessary for splitting string separated by spaces
* str.trim() 🡪 removing whitespace from starting and ending
* str.toUpperCase()
* str.toLowerCase()
* parseInt(str) “42”🡪 42, “42asasas” 🡪 42, “3.14” 🡪 3
* parseFloat(str)
* array.push(int)
* array.pop()
* array.shift() 🡪 pops from the front
* array.unshift(0) 🡪 puts 0 at the front of the array
* array1 = array1.concat(array2)
* array.forEach(funcSignature) 🡪 calls the function for each element as a parameter
* If you want to call something on the class and not on the object, make it a static function.
* Static functions can be called without instantiating any new objects.
* If you have an object in a string that is JSON. Useful for sending data/object.
* const user = JSON.parse(users) 🡪 converts JSON string to JS object.
* const finalString = JSON.stringify(user)
* Date and JSON are not primitive data types but classes that JS provides.
* Object.keys(obj)
* Object.values(obj)
* Object.entries(obj) 🡪 return a nested array of key-value pairs
* Obj.hasOwnProperty(“property”)
* Object.assign({}, obj, {newProperty: “newValue”})

**1.4 Loops, Functions, and Callbacks**

* In anonymous functions, you pass the whole implementation of the function. In the implementation, the function does not have a name. This is why it is called anonymous.
* Anonymous functions and callback functions are related. See example in code.

**1.5 Async, Await, and Promises**

* Synchronous mean only one thing is happening at a time. Sequential.
* Asynchronous mean multiple things are context switching with each other.
* Although JS is single-threaded, it can context switch between multiple things and delegate tasks using async functions.
* setTimeout is an asynchronous function.
* Busy waiting is one way to make setTimeout synchronous.
* The controller delegates the task of setTimeout and when 1s is over, the controller goes on to execute the function itself.
* fs.readFile and Fetch are some common async functions.
* See example of fs.readFile.
* When the thread becomes idle that is when it goes back to the pending callbacks.
* So, even if the callbacks are done, they would wait for the thread to become free. See example of helloworld1 and helloworld2 being printed.
* If our code is synchronous, Call Stack is where we would see all our functions and code.
* JS didn’t have setTimeout, but web introduced it. So, the 5000ms or 5s timer is shown in Web APIs.
* Tasks on Web APIs are tasks that have been delegated. Once the task at Web APIs completes, it goes to Callback Queue and waits.
* When the async function is done (waiting/reading), it will then call the callback function.
* When the thread become idle, it orders Event Loop to check if there is something in the Callback Queue. The Event Loop then takes it and puts it on the Call Stack.
* The async function which finishes first is pushed on to the Callback Queue.
* Callback functions make sense in asynchronous functions.
* Usually, all custom written async functions are wrappers around JS provided async functions like setTimeout or fs.readFile. This is the ugly way, we use promises.
* Promises help us to get rid of callbacks. Callback hell. It is just a class that makes callbacks and async functions slightly more readable.
* In promises, things are the other way around. The controller doesn’t send the callback function. Instead, a promise is returned and when the controller reaches resolve the function passed to .then is called.
* If we have given .then function on a promise, JS resolves that and calls that function.
* The resolve makes sure that the control reaches the callback function. See example.
* A promise can have 3 states. Pending, resolved, rejected.
* If resolve is not executed, logging the promise would show pending. If resolve is inside an async function, then it would not be executed until the very end when the controller is done.
* Whenever we create a promise, we need to pass a function as the first argument which has resolve as the first argument.
* The logic behind promises is that when you have done some asynchronous stuff and completed that is when you call resolve.
* We can use await to get rid of .then. Await lets the delegated task to finish and execute resolve.
* Thread isn’t stuck at await, it is waiting and the same asynchronous stuff is happening under the hood.
* The lines after await wait for the asynchronous stuff to finish. It is conscious decision as we may need the values returned by the async function.
* Makes code much more readable than callbacks/Promises. Usually used on the caller side, not on the side where you define an async function.
* Every await needs to be wrapped inside an async function.