Javascript Notes –

1. **Higher order functions** - functions that take other fn as params or return functions. (foreach,filter, map, reduce)

2. **closures** - function and lexical env - can be used to emulate private function, can be returned and used for working with predefined data.

closure scope chaining - block , function, global

function factory usage to avoid loop problem , anonymous closure, let keyword

3. Javascript is protoype based language ..Inherit properties by following the prototype chain.

4. Javascript is weakly typed language

5. **Prototype chain example**

function Manager() {

Employee.call(this);

this.reports = [];

}

Manager.prototype = Object.create(Employee.prototype);

Manager.prototype.constructor = Manager;

6. **Javascript module pattern** - The Module pattern is used to mimic the concept of classes (since JavaScript doesn’t natively support classes)

so that we can store both public and private methods and variables inside a single object

Common JS - designs and implements JavaScript APIs for declaring modules.

AMD - Asynchronous Module Definition - use define function

define(['myModule', 'myOtherModule'], function(myModule, myOtherModule) {

console.log(myModule.hello());

});

UMD - universal module def

ES6 - introduced built in modules

7**.Call and apply**

.call() is used when the number of the function's arguments are known to the programmer, as they have to be mentioned as arguments in the call statement.

On the other hand, .apply() is used when the number is not known. The function .apply() expects the argument to be an array.

8**. let vs var** -

let: block scope , can use as private members , let variables are not initialized until their definition is evaluated.

The variable is in a "temporal dead zone" from the start of the block until the initialization is processed.(ReferenceError)

var: global or function scope, var are init with undefined

9. **Promise** - A Promise is an object representing the eventual completion or failure of an asynchronous operation.

Promises use ES6 Job queue. It’s a way to execute the result of an async function as soon as possible, rather than being put at the end of the call stack.

Promises are one way to deal with asynchronous code in JavaScript, without writing too many callbacks in your code.

10**. Promise chaining -** promise can be returned to another promise, creating a chain of promises.

Promise rejection at any point in promise chain - This operation will cause the promise chain to skip all the chained promises listed and will skip directly to the catch() statement at the bottom

const status = response => {

if (response.status >= 200 && response.status < 300) {

return Promise.resolve(response)

}

return Promise.reject(new Error(response.statusText))

}

const json = response => response.json()

fetch('/todos.json')

.then(status)

.then(json)

.then(data => {

console.log('Request succeeded with JSON response', data)

})

.catch(error => {

console.log('Request failed', error)

})

Promise.all() - If you need to synchronize different promises, Promise.all() helps you define a list of promises, and execute something when they are all resolved.

Promise.race() - runs as soon as one of the promises you pass to it resolves, and it runs the attached callback just once with the result of the first promise resolved.

11. **Javascript Event Loop** - In general, in most browsers there is an event loop for every browser tab, to make every process isolated and avoid a web page with

infinite loops or heavy processing to block your entire browser. The event loop continuously checks the call stack to see if there’s any function that needs to run.

setTimeout - to defer a function until the stack is clear. Once the timer expires, the callback function is put in the Message Queue.

heap - objects are allocated in heap ( region of memory)

stack - method calls - single thread provided for execution . function calls form stack of frames.

Web APIS - built into browser - They are not part of the JavaScript language itself, rather they are built on top of the core JavaScript language

Web Workers introduce the possibility of parallel execution inside the browser.

12. **Async/await** - since ES2017 asynchronous JavaScript is even simpler with the async/await syntax.

async/await is built on promises.They make the code look like it’s synchronous, but it’s asynchronous and non-blocking behind the scenes.

An async function returns a promise, like in this example:

const doSomethingAsync = () => {

return new Promise(resolve => {

setTimeout(() => resolve('I did something'), 3000)

})

}

When you want to call this function you prepend await

const doSomething = async () => {

console.log(await doSomethingAsync())

}

13. **Javascript anti patterns** -

an anti-pattern is a bad design that is worthy of documenting. Examples of anti-patterns in JavaScript are the following:

* Polluting the global namespace by defining a large number of variables in the global context
* Passing strings rather than functions to either setTimeout or setInterval as this triggers the use of eval() internally.
* Modifying the Object class prototype (this is a particularly bad anti-pattern)
* Using JavaScript in an inline form as this is inflexible
* The use of document.write where native DOM alternatives such as document.createElement are more appropriate