JAVASCRIPT AND REACTJS NOTES BY ANUJ DHINGRA

JAVASCRIPT

**JavaScript** (**JS**) is a lightweight, interpreted, or [just-in-time](https://en.wikipedia.org/wiki/Just-in-time_compilation) compiled programming language with [first-class functions](https://developer.mozilla.org/en-US/docs/Glossary/First-class_Function). While it is most well-known as the scripting language for Web pages, [many non-browser environments](https://en.wikipedia.org/wiki/JavaScript#Other_usage) also use it, such as [Node.js](https://developer.mozilla.org/en-US/docs/Glossary/Node.js), [Apache CouchDB](https://couchdb.apache.org/) and [Adobe Acrobat](https://www.adobe.com/devnet/acrobat/javascript.html). JavaScript is a [prototype-based](https://developer.mozilla.org/en-US/docs/Glossary/Prototype-based_programming), multi-paradigm, single-threaded, dynamic language, supporting object-oriented, imperative, and declarative (e.g. functional programming) styles.

**1) EXECUTION CONTEXT**

**Everything in JS happens inside the execution context.**

Assume execution context to be a big box where everything takes place. It has 2 components in it:

**Memory** : The place where all the variables and functions are stored as (key:value) pairs. Memory component is also known as variable environment.

**Code** : The place where code is executed one line at a time. Code component is also known as Thread of Execution

***JS is a synchronous single-threaded language.***

* By single threaded, we mean JS can only run 1 command at a time
* By synchronous single threaded, we mean it can run 1 command at a time, in a specific order

**2) EXECUTION & CALL STACK**

Everytime you run a program, an execution context is created. When a variable or function is encountered, it is stored in the memory area.

var n=2;

function square(num){

var ans = num\*num;

return ans;

}

var square2 = square(n);

var square4 = square(4);

# **3)** **HOISTING**

**Hoisting** is a concept which enables us to extract values of variables and functions even before initialising/assigning value without getting error

getName(); // in most languages, both lines which are above their declaration will give error. Not in JS though.

console.log(x);

var x = 7;

function getName(){

console.log("Namaste JavaScript");

}

# **4) FUNCTIONS AND VARIABLE ENVIRONMENTS**

# **5) WINDOW AND THIS KEYWORD**

var x = 1;

a();

b(); // we are calling the functions before defining them. This will work properly, as seen in Hoisting (Ep3)

console.log(x);

function a() {

var x = 10;

console.log(x);

}

function b() {

var x = 100;

console.log(x);

}

* Window is a big global object that has a lot of functions and variables. All of these can be accessed from anywhere in the program
* *this* points to *window*
* Global space is anything in JS which isn't inside a function. All these global objects will be present inside the windows schema. But non globals ones won't be there (here, x)
* When a GEC is made, *this* is also created with it (even for functional(local) EC). Global object provided by the browser engine is the window, so *this* points to window.

var a = 10; // not inside any fun. So global object

function b() { // this fun not inside any function. So global.

var x = 5; // not global

}

console.log(window.a); //gives us "a" value

console.log(this.a); //this points to window so it returns "a" value

console.log(a); //also gives same "a" value. (if we dont put any . in front of variable, it \*\*assumes variable is in global space\*\*

console.log(x); // x is not defined. (tries to find x inside global space, but it isn't there)

# **6) UNDEFINED VS NOT DEFINED**

* *undefined* is when memory is allocated for the variable, but no value is assigned yet.
* If an object/variable is not even declared/found in mem alloc phase, and tried to access it then it is *Not defined*

console.log(x); //undefined

var x = 25;

console.log(x); //25

console.log(a); //Uncaught ReferenceError: a is not defined

* **Never** assign *undefined* to a variable manually. Let it happen on it's own accord.

# 7) **SCOPE AND LEXICAL ENVIRONMENT**

* **Lexical Environment** : local memory + lexical env of its parent
* Whenever an EC is created, a Lexical environment(LE) is also created and is referenced in the local EC(in memory space)
* Scope is directly dependent on the lexical environment

function a() {

console.log(b); // surprisingly instead of printing undefined it prints 10.

//So somehow this b could access the b outside the fun.

}

var b = 10;

a();

function a() {

var b = 10;

c();

function c() {

console.log(b); //it prints the right value. How? See ans below Summary part

}

}

a();

console.log(b); // now when cursor comes here, it prints NOT DEFINED!

]

# **8) let, const, temporal dead zone, types of errors**

console.log(a); // ReferenceError: Cannot access 'a' before initialization

console.log(b); // prints undefined as expected

let a = 10;

console.log(a); // 10

var b = 15;

It looks like let isn't hoisted, **but it is**

* Both a and b are actually initialized as *undefined* in hoisting stage. But var b is inside the storage space of GLOBAL, and a is in a separate memory(script), where it can be accessed only after assigning some value to it first.
* ie. one can access 'a' only if it is assigned. Thus, it throws error.
* **Temporal Dead Zone** : Time since when the let variable was hoisted until it is initialized some value.
* So any line till before "let a = 10" is the TDZ for a
* Since a is not accessible on global, its not accessible in *window/this* also

let a = 10;

let a = 100; //this code is rejected upfront as SyntaxError. (duplicate declaration)

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

let a = 10;

var a = 100; // this code also rejected upfront as SyntaxError.

(can't use same name in same scope)

* Let is a stricter version of var. Now, **const** is even more stricter than let.
* -const holds all above properties of let.

let a;

a = 10;

console.log(a) // prints 10 properly. Note declaration and assigning of a is in different lines.

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

const b;

b = 10;

console.log(b); // SyntaxError: Missing initializer in const declaration.

(This type of declaration won't work with const. const b = 10 only will work)

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

const b = 100;

b = 1000;

//this gives us TypeError: Assignment to constant variable.

**TYPES OF ERRORS**

### **Type Error**

The Errors that occur due to conflicts with the declaration type. For example re-assigning const type declaration will throw this.

### **Syntax Error**

The Errors that occur due to wrong syntax that doesn't match with JS Engine syntactical rules.

For example, if const is not initialized, it will throw syntax error as by syntax, it must initialize if it sees a const declaration.

### **Reference Error**

The Errors that occurs if no reference is available for access. Can occur when the variable is no where in scope or maybe it is in temporal dead zone.

# **9) CLOSURES IN JS**

**Closure :** Function bundled together with its lexical environment/scope.

function x() {

var a = 7;

function y() {

console.log(a);

}

return y; // instead of y();

}

var z = x();

console.log(z); // value of z is entire code of function y.

* When y is returned, not only is the fun returned but the entire closure (fun y + its lexical scope) is returned and put inside z. So when z is used somewhere else in program, it still remembers var a inside x()

### **Uses of Closure**

Module Design Pattern, Currying, Functions like once(fun that can be run only once), memoize, maintaining state in async world, setTimeout, iterators...

# **10) setTimeout + Closures**

function x() {

var i = 1;

setTimeout(function() {

console.log(i);

}, 3000);

console.log("This is Hari");

}

x();

***Output***

This is Hari

1 //after waiting 3 seconds (3000ms)

# **11) Callbacks and Event Listeners**

A callback is a function passed as an argument to another function.

setTimeout(function () {

console.log("timer"); // timer

}, 5000);

function x(y) {

console.log("x"); //x

y();

}

x(function y() {

console.log("y"); //y

});

* In the call stack, first x and y are present. After completion, they go away and stack is empty. Then after 5 seconds(from beginning) anonymous suddenly pops up in stack ie. setTimeout
* All 3 functions are executed through call stack. If any operation blocks the call stack, its called **blocking the main thread**
* Say if x() takes 30 sec to run, then JS has to wait for it to finish as it has only 1 call stack/1 main thread. *Never block main thread*.
* **Always use async for functions that take time eg. setTimeout**

**Event Listener**

* When we create a button in HTML and attack a clickListener in JS :
* in index.html

#### **Garbage Collection and removeEventListeners**

* Event listeners are heavy as they form closures. So even when call stack is empty, EventListener won't free up memory allocated to count as it doesn't know when it may need count again.
* **So we remove event listeners when we don't need them (garbage collected)**
* onClick, onHover, onScroll all in a page can slow it down heavily.

# **16) Asynchronous JS and Event Loops**

Asynchronous means that things can happen independently of the main program flow.In the current consumer computers, every program runs for a specific time slot and then it stops its execution to let another program continue their execution. This thing runs in a cycle so fast that it's impossible to notice. We think our computers run many programs simultaneously, but this is an illusion (except on multiprocessor machines).

* Event loop checks the callback queue, and if it has element puts it into call stack. It is a *gate keeper*.

<button id="clickMe">Click Me!</button>

in index.js

document.getElementById("clickMe").addEventListener("click", function xyz(){ //when event click occurs, this callback function is called into callstack

console.log("Button clicked");

});

**REACTJS NOTES (BASIC)**

* React is a js (javascript) library for building user interfaces (UIs). It lets you build complex UIs by combining isolated pieces of code called components.

import React from 'react';

import ReactDOM from 'react-dom';

import './index.css';

* If your entire application is contained within a component named Board (for example), here's how you mount that component onto the DOM. To do so, enter this line at the bottom of your file.
* ReactDOM.render (<Board />, document.getElementById ('root'));
* There are a few types of components. We start with React.Component.
* A React Component Class (or React Component Type) takes in parameters (called props) and returns a hierarchy of views to display via the render method.
* The render method returns a description of the view you want to show. This description returned by render is a React element.

class ShoppingList extends React.Component {

render () {

return (

// Contents go here

);

}

}

* The contents of a React element can be written in plain js or in JSX, which is a XML-like syntax that is automatically converted to plain js. This JSX:

<div className="shopping-list">

<h1>Shopping List for {this.props.name}</h1>

<ul>

<li>Instagram</li>

</ul>

</div>;

* Is converted by React to this js:

React.createElement ('div', {className: 'shopping-list'},

React.createElement ('h1', null, 'Shopping List for ', this.props.name),

React.createElement ('ul', null,

React.createElement ('li', null, 'Instagram')

)

);

* To add event handlers to allow user interaction, React also uses properties. For example, onClick:

<button onClick={function () {alert ('click');}} />

* Or we can write the same with the arrow function format for writing functions in js ES6. Writing functions like this has the advantage in that the value of this is more amenable to object-oriented programming.
* When inspecting the generated button, the onClick doesn't appear like a normal onclick event handler bound to the DOM element. The event is handled by React internally and is not accessible from the DOM.
* The state of the program is held within the component, in the component's state. This state can be accessed by invoking this.state.
* To initialize the state of the component, here's the invocation you should enter the following code. This code in particular will set the state to {value: null}:

constructor (props) {

super (props);

this.state = {value: null};

}

* In the invocation above, the only changeable part is what goes within this.state. The call to super is mandatory and must be done at the top of the constructor.
* To update the state of the component, use the function this.setState ({...}). This can also be done from within a DOM element within the component. For example, this code will set this.props.value to 'X' when the element is clicked:
* <button onClick={() => this.setState ({value: 'X'})} />
* After clicking on said button, this.state.value will be set to X. To see it in action, let's place this.state.value as the contents of the button. When you click on the button for the first time, this.state.value will change and you'll see an 'X' within the button.

class Square extends React.Component {

constructor (props) {

super (props);

this.state = {value: null};

}

render () {

return (

<button onClick={() => this.setState ({value: 'X'})}>

{this.state.value}

</button>

);

}

}

## [**Main concepts**](https://reactjs.org/docs/hello-world.html)

* JSX makes you close void tags (or non-void tags that happen to have no content) by adding a trailing slash before closing the tab:
* const element = <img src={user.avatarUrl} />;
* Normally, React is rendered within a DOM element with id root. However, you can make React render more than one component at the same time, as long as they belong to different DOM elements.
* React elements are immutable. To change things, you need to create them anew and re-render. Interestingly enough, you can call ReactDOM.render multiple times.

function tick () {

const element = (

<div>

<h1>Hello, world!</h1>

<h2>It is {new Date ().toLocaleTimeString ()}.</h2>

</div>

);

ReactDOM.render (element, document.getElementById ('root'));

}

setInterval (tick, 1000);

* All React components must act like pure functions with respect to their props."