JS Notes

* Undefined vs null
  + undefined is the default value of a variable that has not been assigned a specific value. Or a function that has no **explicit** return value ex. console.log (1). Or a property that does not exist in an object. The JavaScript engine does this for us the **assigning** of undefined value.
  + null is **"a value that represents no value"**. null is value that has been **explicitly** defined to a variable.
* && Operator
  + The && or **Logical AND** operator finds the first *falsy* expression in its operands and returns it and if it does not find any *falsy* expression it returns the last expression
    - firstVariable = 0, secondVar = 1
    - firstVariable && secondVar // return false as first is false
* || operator
  + The || or **Logical OR** operator finds the first *truthy* expression in its operands and returns it.
    - console.log (null || 1 || undefined) // logs 1
* + unary operator
  + The + unary is the fastest way of converting a string to number
* DOM
  + DOM (Document object model): an interface API for HTML and XML documents. When the browser first reads HTML, it created big object (DOM). It is a tree like structure and is used for interacting and modifying the DOM structure or specific elements or nodes
* Event Propagation
  + Event on DOM doesn’t entirely occur on just that element.
  + **Bubbling phase (Event Bubbling)**, the event bubbles up or it goes to its parents, grandparents until reach to window.
  + **Capturing phase (Event Capturing):** the events start from the window down to element that triggered the event or the event.target.
  + **Event phases**
    - **Capturing phase:** window to target element
    - **Target phase:** event reached the target
    - **Bubbling phase:** Event bubbles up from element to window
* Event Bubbling and Capturing
  + the **addEventListener** method has a third parameter **useCapture default false**
  + in False state, event occur in bubbling phase
  + in true state, event occur in capturing phase
* event.PreventDefault and event.stopPropagation
  + **preventDefault**: prevents the default behaviour of an element. If used in a form it prevents form submitting and with anchor it prevents from navigating.
  + **stopPropogation:** it stops the propagation of an event or it stops the event from occurring in bubbling or capturing phase.
* Find if the **event.preventDefault** method used in an element
  + event.defaultPrevented property in the event object, will returns Boolean if the event.preventDefault was called
* why following code throws an error  
  const obj = {}  
  console.log (obk.someprop.x )
  + someprop is undefined and code is trying to access “**x**” property of undefined
* event.target
  + it is the element on which the event occurred or triggered the event
* event.currentTarget
  + it is the element on which we attach the event handler explicitly
    - <div onClick=”clickFunc(event)”> button </div>
    - Fun clickFunc => console.log(event.currentTarget)
      * Log the div: attach the event handler
* == vs ===
  + == (abstract equality): compares by value after coercion
  + === (strict equality): compares by value and type without coercion
  + Coercion: a process of converting a value to another type.
* Why false  
  let a = { a: 1 }, b = { a: 1};  
  let c = a;  
  console.log( a === b) // log false even have same  
  console.log( a === c) // log true
  + JS compares objects and primitives differently.
  + **Primitives** compares by values
    - String, number, null, undefined, Boolean, symbol, bigint
  + **Objects** compares by references or the address in memory where the variable is stored
* **!! operator**
  + It’s a fancy way to converting a value into a Boolean
* Multi expression in one line
  + Comma operator to evaluate multiple expression in one line.
  + It evaluates from left to right and returns the value of the last item on the right or the last operand.
  + Example  
    let x = 2;  
    x = ( x++, x\*= 2, x+=10)  
    **x** will be 16
* **H**oisting
  + Term is used to describe the moving of variables and functions to the top of their scope on where we define that variable or function
  + Execution context: It is the environment of code that is currently executing
  + Execution content has two phases: compilation and execution
  + Compilation: it gets all the function declaration and hoists them up to the top of their scope, so we can reference them later and gets all variable declaration (declared with var) and also hoists them up and give them a default value undefined
  + Execution**:** In this phase, it assigns value to the variables hoisted earlier and executes
* Scope
  + Scope is area where we have valid access to variables or function
  + Three scopes
    - Global: function or variables declared in the global namespace are in the global scope and therefore is accessible everywhere in our code
    - Function scope: All are declared within a function are not accessible outside of it
    - Block scope: let , const declared within a block can only be access within it.
* **Lexical** scope
  + **A**bility of a function scope to access variables from the parent scope
  + **S**cope is a set of rules for finding variables. If a variables doesn’t exist in the current scope, it look ups and search in the outer scope and if doesn’t find, it reaches the global scope if the variable not exists, it throws an error.
  + It is also called scope chain
* Closure
  + It is simply the ability of a function at the time of declaration to remember the references of variables and parameters on its current scope, on its parent function scope, until it reaches the global scope with the help of scope chain.
  + Basically it is the scope created when the function was declared
  + A closure is the combination of a function bundled together with the references to its surrounding state.
* Falsy values
  + “” // empty string
  + 0
  + **Null**
  + Undefined
  + Nan
  + False
* Check if a value is falsy
  + **!!NaN double not operator**
* **Use strict**
  + **“use strict”** is a ES5 feature in JavaScript, that makes our code in strict mode in function or entire scripts
  + **Restrictions**
    - Accessing or assigning a variable that is not declared
    - Assigning a value to a read-only or non-writable global variable
    - Deleting an undeletable property
    - Duplicate parameter values
    - Creating variables with the use of the eval function
    - Default value of “**this**” will be undefined
* this in JavaScript
  + **‘this’** refers to the value of the object that is currently executing or invoking the function.
  + **this** value changes depending on the context on which we use it and where.
    - **Ex**ample  
      const carDetails = {  
      name: ‘Ford’,  
      getName: function(){  
       console.log(this.name)  
      }  
      };carDetails.getName()) // logs Ford
    - var name = ‘Audi’  
      var getName = carDetails.getName // method assigned to variable  
      getName() // logs Audi
    - Reason: getName method has a different “owner” object that is the window object.
    - Declaring var keywords in the global scope attaches properties in the window object with the same name as the variables.
    - Remember, this in the global scope referes to the window object when “use strict” is not used.
    - **Solve above issue**
      * It can be solved with apply and call methods in function
      * getName.apply(carDetails) // log Ford
      * getName.call(carDetails) // log Ford
      * apply and call expects the first param to be an object which would be value of “this” inside function
    - IIFE (immediately invoked function expression), global scope functions, Anonymous function and inner function in methods inside an object has default this points to window
    - Solve above issue one way  
      const carDetails = {  
      getName: function(){  
      const self = this; // saves the this value to self  
       console.log(self.name)  
      }  
      };
    - **ES6 arrow functions**
      * **Arrow** functions does not have its own this. It copies the value of this of the enclosing lexical scope.
      * const carDetails = {  
        name: ‘Ford’,  
        getName: () => { console.log(this.name }  
        };
* prototype of an object
  + Is a blueprint of an object and used as a fallback for properties and methods if it doesn’t exist in the current object.
  + It’s the core concept around JavaScript’s prototypal inheritance
  + Example  
    const o = {};  
    console.log(o.toString()); // logs [object Object]  
    toString doesn’t exist in the object it doesn’t throw error, instead returns a string.
  + When a property doesn’t exist in the object it looks into its prototype and if it doesn’t exist it looks into the prototype’s prototype and finds it with the same in Prototype chain.
* IIFE (Immediately invoked function expression)
  + It’s a function that is get invoked or executed after its creation or declaration.
  + Syntax
    - Wrap the function inside a parenthesis.
      * **(function(){}());**
      * **(function(){})();**
      * **(function named(params){}());**
* Function.prototype.apply method
  + Apply invokes a function specifying the “this” or the “owner” object of that function on that time of invocation.
  + Example:  
    **const** details = {  
     message: ‘Hello World!’  
    }; **function** getMessage(){  
     return this.message;  
    }  
      
    getMessage.apply(details); // logs: Hello World
* Function.prototype.call method
  + Apply invokes a function specifying the “this” or the “owner” object of that function on that time of invocation.
  + Example:  
    **const** details = { message: ‘Hello World!’ }; **function** getMessage(msg){  
     return` ${msg} ${this.message}`;  
    }  
      
    getMessage.apply(details, [‘Call ’]); // logs: Call Hello World
  + Difference is how we pass arguments
* **Function.prototype.bind**
  + Bind method returns a new function that is bound to a specific this value or the owner project object.
  + The returned new function can be used later.
  + Call, apply methods invoke the function immediately
  + Bind returns a new function to bind
    - Example  
      class MyComponent extends React.Component {  
       constructor(props)  
       {  
       super(props);  
       this.state = {  
       value: “”  
       };  
       this.handleChange = this.handleChange.bind(this)  
       }  
         
       handleChange(e) { // do somethind }  
         
       render(){  
       return (  
       <>  
       <input onChange={this.handleChange} />  
       </>  
       )  
       }  
      }
    - Above the.handleChange holds the function returns from bind
* Functional programming
  + It is a declarative programming paradigm or pattern on how we build our applications with functions using expressions that calculates a value without mutating or changing the arguments that are passed.
  + Example
    - Array has map, filter and reduce methods which are the most famous because of their usefulness and they don’t mutate or change the array
  + JavaScript support closures and high order functions which are a characteristic of a functional programming language
* HighOrderFunctions
  + These are functions that can return a function or receive arguments which have a value of a function
  + Example  
    function highOrderFunction(param, callback){  
     return callback(param);  
    }
* **Why** functions are called First-class Object.
  + Because functions are treated as any other value in the language.
  + They can be assigned to variables.
  + They can be properties of an object which are called methods
  + They can be an item in array
  + They can be returned as values of a function
* Array.prototype.map method example by hand
  + Ex:  
    function map(arr, mapCallBack){  
     if( !Array.isArray(arr) || !arr.length || !typeOf mapCallBack !== ‘function’ ){  
     returns [];  
     }  
       
     let result = [];  
     for(let i = 0; i < arr.length ; i++){  
     result.push( mapCallBack(arr[i], i, arr) );  
     // push the result of the mapCallback in the result array  
     }  
     return result;  
    }
* Array.prototype.filter method example by hand
  + Ex:  
    function filter(arr, filterCallBack){  
     if( !Array.isArray(arr) || !arr.length || !typeOf mapCallBack !== ‘function’ ){  
     returns [];  
     }  
       
     let result = [];  
     for(let i = 0; i < arr.length ; i++){  
     if(filterCallBack(arr[i], i, arr)){  
     result.push(arr[i]);  
     }  
     }  
     return result;  
    }
* Array.prototype.reduce method example by hand
  + Ex:  
    function reduce(arr, reduceCallBack, initialValue){  
     if( !Array.isArray(arr) || !arr.length || !typeOf mapCallBack !== ‘function’ ){  
     returns [];  
     }  
       
     let hasInitialvalue = initialValue !== undefined;  
     let value = hasInitialValue ? initialValue : arr[0];  
     let i = hasInitiateValue ? 0 : 1;  
     for(i; i < arr.length; i++){  
     value = reduceCallBack(value, arr[i[, i, arr);  
     }  
     return value;  
    }
* **arguments object**
  + It is a collection of params value pass in a function
  + It’s an array-like object but doesn’t have built-in method in an array forEach, reduce, filter and map.
  + arguments can be convert into array
    - Array.prototype.slice.call(arguments)
  + arguments does not work on ES6 array functions.
  + It can be solved if environment supports the rest syntax.
    - Const four = (…args) => args;
* Create object without a prototype
  + Object.create helps to create method without a prototype.
  + Ex:
    - Const o2 = Object.create(null);
* Why does b in this code become a global variable when you call this function.
  + Ex.  
    function myFunc()  
    {  
     let a = b = 0;  
    }  
    myFunc()
  + Reason is the assignment operator ( = ) has right to left associativity.
    - It becomes as let a = (b=0);
    - b = 0, is not declared while evaluation, so JS engine makes a global variable b outside this function after the return value of expression
    - Solution
      * Let a,b;  
        a = b = 0;