3/3/24. JS

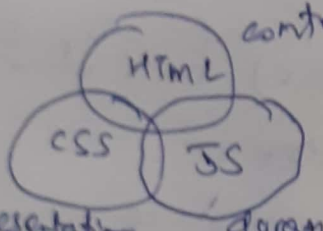
1) console.log(' ') → writes o/p to the console & it is very for debugging.

2) JS
i) lightweight (doesn't eat computers memory & has an easy syn
ii) cross-platform (can be used on multiple platforms, not only as web developer)
iii) object-oriented

3) Client side → traditionally used
server side → node.js.

4) Dynamic effects and interactivity.

5) Core Technologies of web Output



HTML → Nouns →
CSS → ~~Presentation~~ Adjectives → pf color: red;
JS → verbs → p.hide()

6) Versions

ES5 → ES6 / ES 2015 → ES7 / ES 2016 → ES8 / ES 2017

7) Variables & Data Types.

var fname = 'John';

a/p :- dynamic typing

Data Types

Numbers → decimals & integers
Strings → sequence of characters.
Boolean → logical data type that can be true or false
Undefined → DT of a var. that doesn't have a value yet
Null → means 'non-existent'

8) var job;
console.log(job);
>> undefined

9) note :-
var name cannot start with number.

10) Comments :-
// (single line) /* .... */ (multi line)

11) Type - coercion
var name = 'John';
var age = 25
console.log(name + ' ' + age);
string string
JS automatically transforms int this string & creates a big string. This process is known as type coercion.

12) Variable - mutation
var age = 25
age = 'eight-teen'; } can be reassigned to data type of other
in later parts above.

13) alert :- used to make sure info. comes through to user (comes with an OK button)

14) prompt :- to ask for user-input . (comes with OK & cancel button).
var a = prompt ("Enter his unit name");
console.log(a);

15) Operators
math operators +, -, *, /
comparison logical operators <, >, ==, ===
var a = 18 > 30
console.log(a)

16) typeof operator
console.log(typeof name)

(16) var x, y; // multiple assignment
x = y = 26;
→ (Assignment operator works from right to left which is anomalous behaviour)

(17) ~~Is control~~

(18) Equality operators.

==	===	!=	!==
(equal val)	(equal val. & equal type)	(not equal)	(not equal value (or) not equal type)

(18) Is control

```
if (age < 18)
{
  y
else
  y
```

(19) Basic Boolean logic :-

NOT	AND &	OR
(!)	(i.e &&)	(||)
→ (inverts true/ false value)	→ true if All are true	→ true if one is true

→ if (age > 20 && age < 30)

(20) Teeray operator :-

a) var age = 10;

age >= 20 ? console.log ("true") : console.log ("false")

b) var drink = age < 18 ? 'mango' : 'orage'

(21) Switch.

```
var job = 'teacher'
switch (job) {
    case 'teacher':
        console.log ("teaches");
        break
    case 'driver':
        break ("driver");
    default:
        console.log ("does something else")
}
```

Note:- Having multiple values for a case.

```
switch(job){
    case 1:
    case 2:
    case 3: console.log(x);
    case 4:
}
```

for 1, 2, 3, same thing will get executed.

(22) Falsy values :- undefined, null, 0, '', NaN.

converted to false in if-else auth.

Truthy values :- NOT falsy values.

(23) functions

```
function calculateAge(birthYear){
    return 2024 - birthYear;
}
var x = calculateAge(2000)
console.log(x)
```

(24) Difference b/w function declaration & function expression.

```
var calculateAge =
    function(birthYear){
    }
```

// function calls remain the same.

→ If we use switch, then in place of break, we use return.

(25) Arrays.

var names = ['John', 'Mark', 'Jane']; // Most common form of array creation

var years = new Array (1900, 1901, 1905)

console.log(names[0]);

console.log(naes.length);

names[1] = 'Ben';

console.log(naes[1]);

Arrays: Different data types:

var john = ['John', 'Smith', 1990, 'teacher', false]

john.push('blue') // adds to the last ele.

john.unshift('Mr.'); // Adds as a first ele in array.

john.pop() // removes the last ele.

john.shift() // " " first ele.

john.indexOf(1990) // returns index of perticular ele

john.indexOf('designer') === -1 ? 'John is NOT a designer' : 'John is a designer';

// indexOf returns -1, if ele is not present in array.

(26) Objects & properties

(key-value pairs)

-> var john = {

firstName: 'John', lastName: 'Michael', year: 1990,

family: ['Jake', 'Hector', 'Paula'],

job: 'teacher', isMarried: false };

Note:

=> john is an object with firstName, lastName, family as properties

~~var john~~ = console.log(john);

→ Accessing members :-

console.log(john.firstName); ✓

console.log(john['lastName']); ✓

→ We can also change values :- john.age = 32 ✓

* As arrays, objects can also be created using new operator.

var john = new Object();
john.fname = 'Jackie';
:

(27) Objects & Methods.

var john = {
name: 'Jackie',
byear: 1990
age: function(byear) → we can refer it in this keyword, i.e it refers to the current object.
{ return 2024 - byear
}
this.byear
}
console.log(john.age(1990)) ↳ & pass no parameter i.e function()

Append the result to object by :- john.result = john.age(1990)

At we can also create a new property like :-

→ this.result = 2024 - this.byear;

(28) Loops and Iteration

i) for (var i = 0; i < 10; i++)
{ console.log(i)
}

ii) while (i < 10) {
3

Note :- if (typeof 'John' + 'J' !== 'String')
↳ returns the type of the particular variable

01/02/2021 (1996)

1) JS → changed from LiveScript to JS to attract Java developers

2) ES 1 (EcmaScript 1) → became the first version of JS was standard.

→ ECMAScript → (The language standard)
→ JS (The language in practice)

3) 2009 ES5 :- released with lots of new features

4) 2015 ES 6 / ES 2015 :- Biggest update to the language ever

5) 2015 :- Changed to annual release cycle.

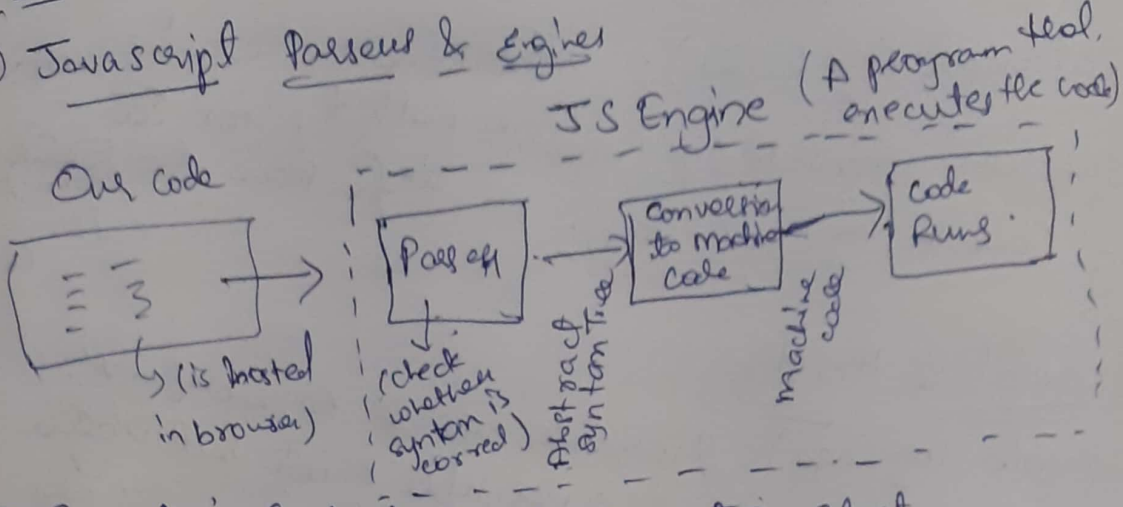
6) 2016, 2017, 2018, 2019 ... Release of ES 2016 / ES 2017 / ES 2018 / ES 2019 / ...

Which version to use?

- ES 5 → Fully supported in all browsers / Ready to be used today
- ES 6 / ES 7 / ES 8 → well supported in all modern browsers
  ↳ No support in older browsers.
  ↳ can use most features in production with transpiling & polyfilling (converts to ES 5)

How JS works behind the scene

① Javascript Parsers & Engines



② Execution Contexts & the Execution Stack.

i) ↳ (A box (or) a wrapper which stores variables and in which a piece of code is evaluated & executed).

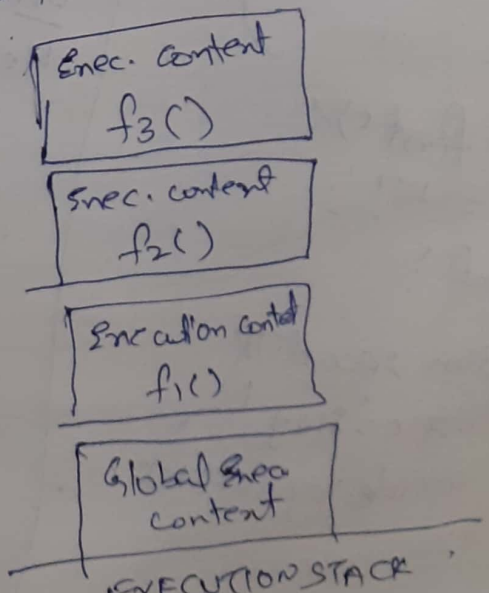
ii) The default execution context is Global Execution Context:

↳ code that isn't inside any function / associated with global object (In the browser, that's the window object (eg:- lastName === window.lastName (True).

iii) Execution Stack:-

-> Each time, we call a function, it gets its own execution context

```
var age = 'John';
function f1() {
  f2();
}
function f2() {
  f3();
}
function f3() {
  f1();
}
```



③ Function Hoisting

```
i) calculateAge(1962);
function calculateAge(year){
    console.log(2016-year);
}
```

// function can be called prior to the function declaration

④ Variable Hoisting

```
console.log(age); // variables are set to undefined i.e variable exists but value not assigned.
var age = 13
```

Scoping in JS.

1) Each new function, creates a scope (the space/environment, in which the variables it defines are accessible).

2) Lexical scoping :- A function i.e is written inside another function, gets access to scope of outer function.

Eg:-

```
var a = 'Hello';
first();
function first(){
    var b = 'Hi!';
    second();
    function second(){
        var c = 'Hey!';
        console.log(a+b+c);
    }
}
```

Global variable a can be accessed from anywhere

O/P:-
Hello Hi Hey!

The this variable.

→ Each & every execution context, gets this variable.

→ Regular function call :-

⇒ "this" keyword points at global object i.e window object in browser.

→ method call :-

"this" variable points to the object i.e is calling the method.

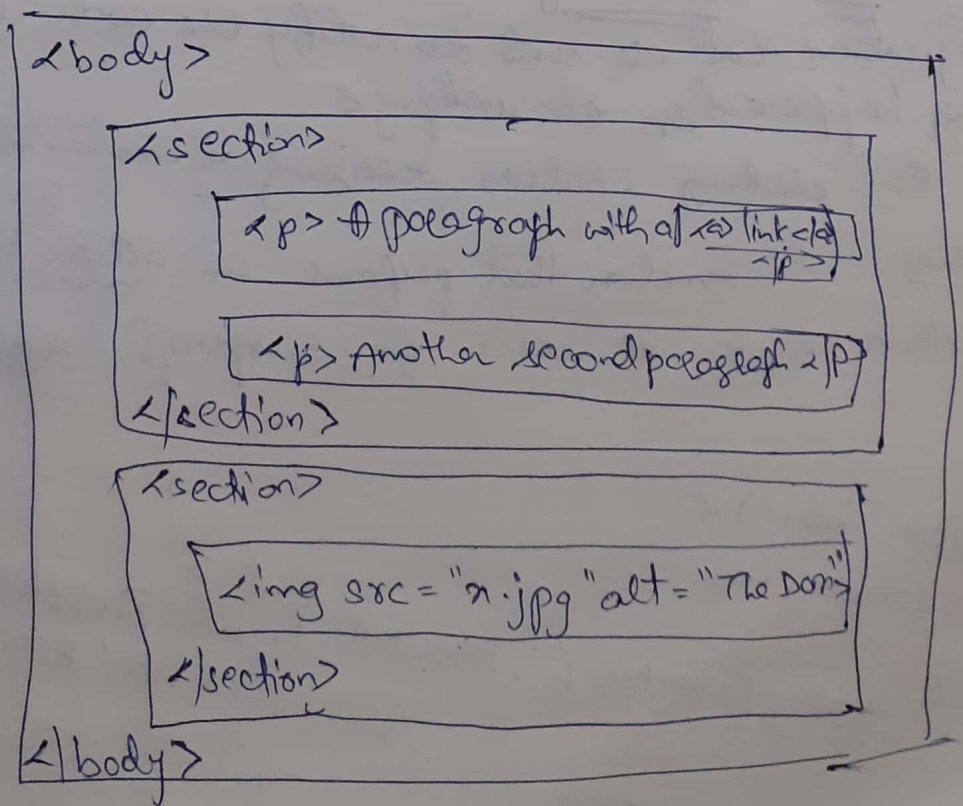
The DOM and DOM object model.

i) Dom refers to the Document object model.

ii) Structured reps. of an HTML doc.

iii) The Dom is used to connect webpages to scripts like JS.

iv) For each HTML box, there is an object in Dom, that we can access & interact with.



Html and CSS recap

DOM Access & Manipulation

i) document.querySelector('#score-0').textContent;
↳ (To select elements from the webpage)
↳ (refers to id or class)

( )

ii) docuet.querySelector('#score-0').
innerHTML = '' + dice + '';
↳ we can pass on Html content too.

iii)
var x = docuet.querySelector('#score-0').textContent

iv) To change styling of dice class:-

document.querySelector('.dice').style.display =
'none';

Events & Event Handling

↳ Notifications that are sent to notify the code, that something happened on the webpage.

Ex:- clicking a button, resizing a window, etc.

Event Listener:- A function that performs an action based on a certain event. It waits for a specific event to happen

Ex:-
function btn() {
// do-something here
}
document.querySelector('.btn_roll').addEventListener('click', btn

(or)

//we can use an anonymous function.

- document.querySelector('.btn-roll').addEventListener(
'click', function(){

});

≡

document.getElementById('score-0').textContent = '0';

↳ (faster than query selector)

---

state variable :- to remember the state of system.

var gamePlaying = true;

Everything is an object: Inheritance & the Prototype chain

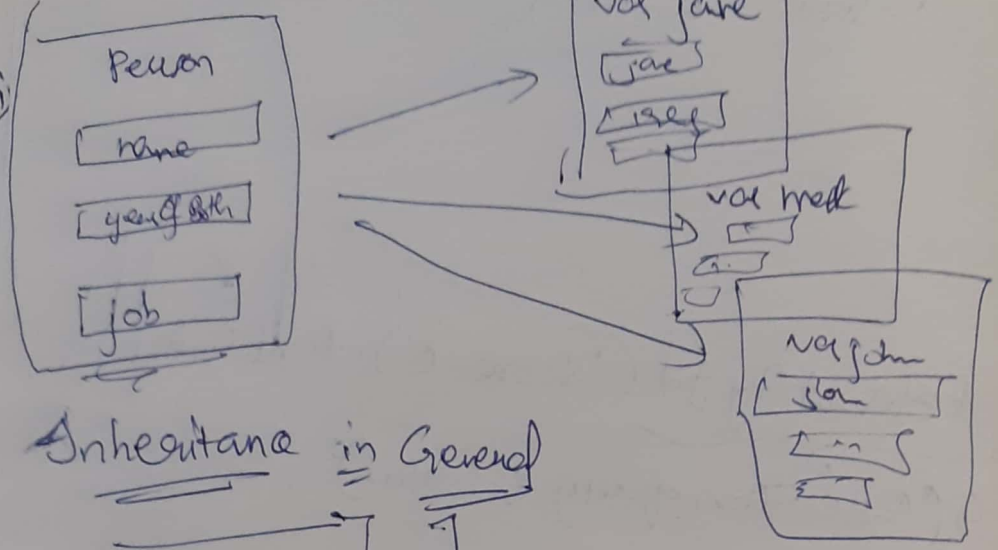
Primitives	Everything else ...
Number	Arrays
String	Functions
Boolean	Objects
Undefined	Dates ...
Null	↳ an object

OOP

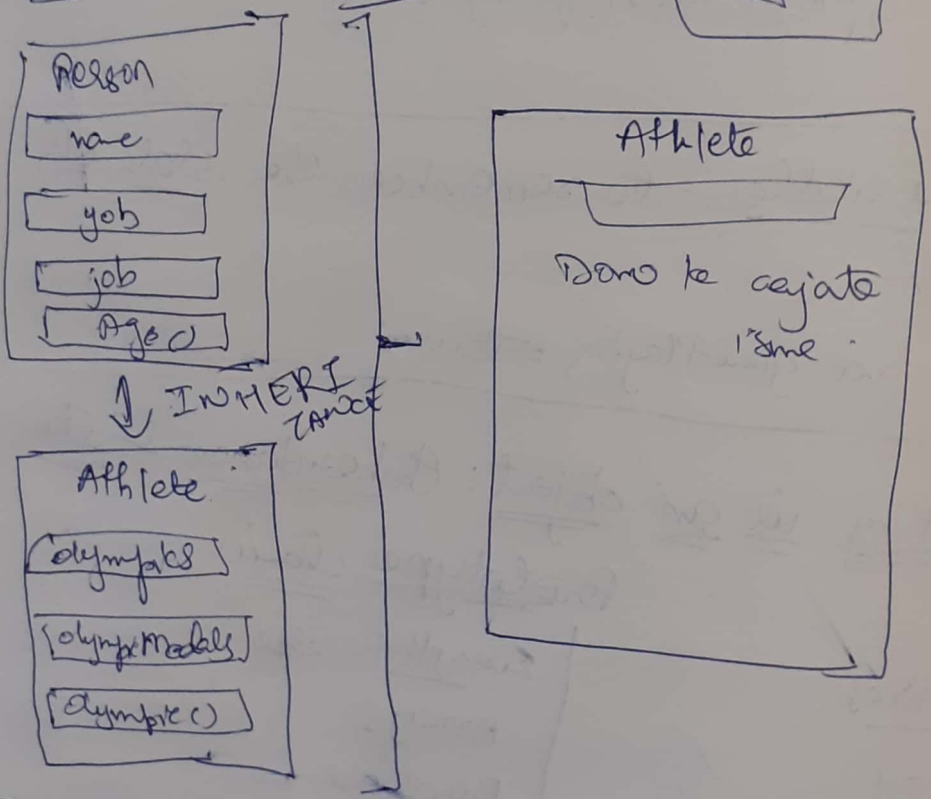
1) objects interacting with one another through methods & props.
2) used to store data, structure efficiently into modules

Constructor in JS

Constructor



Inheritance in General



** Everything in JS inherits from Object.

1) Every JS object has a prototype property, which makes inheritance possible in JS.

2) The prototype prop of an object is where we put methods & props, that we want other objects to inherit.

new operator :- a brand new empty object is created, & then the function is executed

Function Constructor

```
var Person = function (name, yob, job)
{
    this.name = name;
    this.yob = yob;
    this.job = job;
}

var John = new Person ('John', 1980, 'teacher')
                //creating a John object

this.calculateAge = function()
{
    console.log(2016 - this.yob);
}

john.calculateAge()
```

→ Similarly, we can use the prototype property, outside the Person function :-

```
Person.prototype.calculateAge = function(){
    =
};
```

→ john instanceof Person // true in console

note :-

In console :-

```
>> var x = [1,4,8]
>> console.dir(x)
```

(in the result click on prototype property we find all the methods that can be used with x)

OBJECT.CREATE

→ First, we create an object prototype & from that we create the other object directly (inherits).

```
var person = {
    calAge : function() {
    }
};
var john = Object.create(person);
john.name = 'mak'
john.age = 18
```

(or)

```
var john = Object.create(person,
    {
      name: 'mak', age: 18
    })
```

→ In Function constructor, the newly created ~~constructor~~ object inherits from the constructor prototype property.

04/04/21

Primitives (PS)

→ Vars containing primitives ~~they~~ they ~~vars~~ are ~~stored~~ inside of the var

Ex:-
```
var a = 30
var b = a
a = 46
console.log(a) //46
console.log(b) //30
```

Objects

→ Vars associated with objects do not contain the object, but they ~~do~~ contain a reference to a place in memory, where object is stored

→ Var declared as object doesn't have real copy of object, it just points to the object.

for objects

```
var obj1 = {
    name: 'John',
    age: 29
}
var obj2 = obj1;
obj1.age = 37
console.log(obj1.age) //37
console.log(obj2.age) //37
```

FIRST-CLASS FUNCTIONS

1. A function is an instance of ~~Object~~ Object type.
2. We can store functions in a variable.
3. We can pass a function as an argument to another function.
4. We can return a function from a function.

17:12

* //Passing functions as arguments

```
var years = [1990, 1965, 1937, 2005, 1998];
function arrayCalc(arr, fn) {
    var arrRes = [];
    for (var i = 0; i < arr.length; i++) {
        arrRes.push(fn(arr[i]));
    }
    return arrRes;
}
function calculateAge(el) {
    return 2016 - el;
}
var ages = arrayCalc(years, calculateAge)
console.log(ages)
```

// Returning functions

```
function interviewQuestion(job) {
  if (job === 'designer') {
    return function(name) {
      console.log(name + ', explain UX design');
    }
  } else if (job === 'teacher') {
    return function(name) {
      console.log('what subject do you teach, ' +
        name + '?');
    }
  } else {
    return function(name) {
      console.log('Hello ' + name + ', what do
        you do?');
    }
  }
}

var t1 = interviewQuestion('teacher'); // without storing it in variable we can write the above line as:
t1('John'); // what subject do you teach, John?

interviewQuestion('teacher')('John')
// evaluated from left to right.
```

initially interviewQuestion('teacher') is executed & it returns a function & then we call that function by passing 'John' as parameter

Immediately Invoked function Expression (IIFE)

→ If the purpose is to hide a variable (i.e is to keep it as private) by declaring it inside a function, we can do IIFE.

```
function game(){
    var score = Math.random() * 10;
    console.log(score >= 5);
}
game();
```

(or) use IIFE

```
(function(){
    var score = Math.random() * 10;
    console.log(score >= 5);
})();
```

↳ To tell JS that it is an expression. & also score cannot be accessed outside.

→ We can also have parameters in IIFE.

→ IIFE can only be called once.

→ IIFE's main goal is data privacy.

Closures

i) One of the most crucial & advanced & difficult thing in JS.

ii) An inner function has always access to variables & parameters of its outer function even after the outer function has returned.

- P.T.O

```
function retirement(retAge){
  var a = 'years left until retirement';
  return function(yOB){
      var age = 2016 - yOB;
      console.log((retAge - age) + a);
  }
}

var a = retirement(66)
a(1990); // 40 years left until retirement.
```

Bind, call and apply

→ call() → It calls a function with given this value and ~~plus~~ arguments provided individually.

Ex:-
```
var john = {
    name: 'John'
    age: ..
    x:
    y:
    presentation: function(a, b)
        console.log(... + this.age +
            ... + this.x ... + b);
};
john.presentation(10, 30);

var emily = {
    name: 'Emily',
    age: 35
    job: 'designer'
};
// emily does not have presentation function;
but using john object we can call-ek
```

presentation function on emily.

john.presentation.call(emily, 30, 40)
passing parameters to the function
It takes value of this variable

//All the "this" instances in presentation, will correspond to the emily object.

→ bind()

- similar to call.
- It doesn't call the function immediately, but instead generates a copy of function, so we can store it.

Ex:-

var johnF = john.presentation.bind(john, 10)
this variable value for parameter a is set.

johnF(30) //value for parameter b can be set now & function is invoked.

johnF(20) // can be done many a times, since function is stored.

var emilyF = john.presentation.bind(emily, 10).

emilyF(30) ✓

---

new:- It creates an empty function.

Callback function

→ A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to

complete same kind of routine (or) action.

Ex:- function greeting (name) {
alert ('Hello' + name);
}
function processUserInput (callback) {
var name = prompt ('Please enter the name')
callback (name);
}

processUserInput (greeting)

→ Callback functions are used in asynchronous functions, where one function has to wait for another functio (like waiting for a file to load).

Note:- Do a budget App.

Step1:- Project planning & Arch.

To-Do List

i) Add event handler
ii) Get the values
iii) Add the new item to DS
iv) Add new item to UI
v) Calculate budget
vi) Update the UI

→ Transforming the To Do list to modules

UI module
i
ii
iv
v

Data module
iii
v

Controller module
i)

Structuring our code with modules

→ keeps the units of code for a project both cleanly separated & organized.

→ Encapsulate the data into privacy & expose only data publicly.

Implementing the Module Pattern

→ The main aim of module pattern is that it returns an object containing all the functions that we want to be public.

```
var budgetController = (function() {
    var x = 23;
    var add = function(a) {
        return x + a;
    }
    return {
        publicTest: function(b) {
            console.log(add(b));
        }
    }
})();

budgetController.x // cannot access
" .add(5) // " "
budgetController.publicTest(5) // 28
```

→ Because of closures, inner function has access to variables & functions of outer function

Setting up Event listener

→ document.querySelector('.add__btn').addEventListener
('click', function() {
console.log('Button was clicked');
});

→ Now using the event object :-

document.addEventListener('keypress', function(event) {
console.log(event);
if (event.keyCode === 13) {
console.log('Enter was pressed.');
}
});

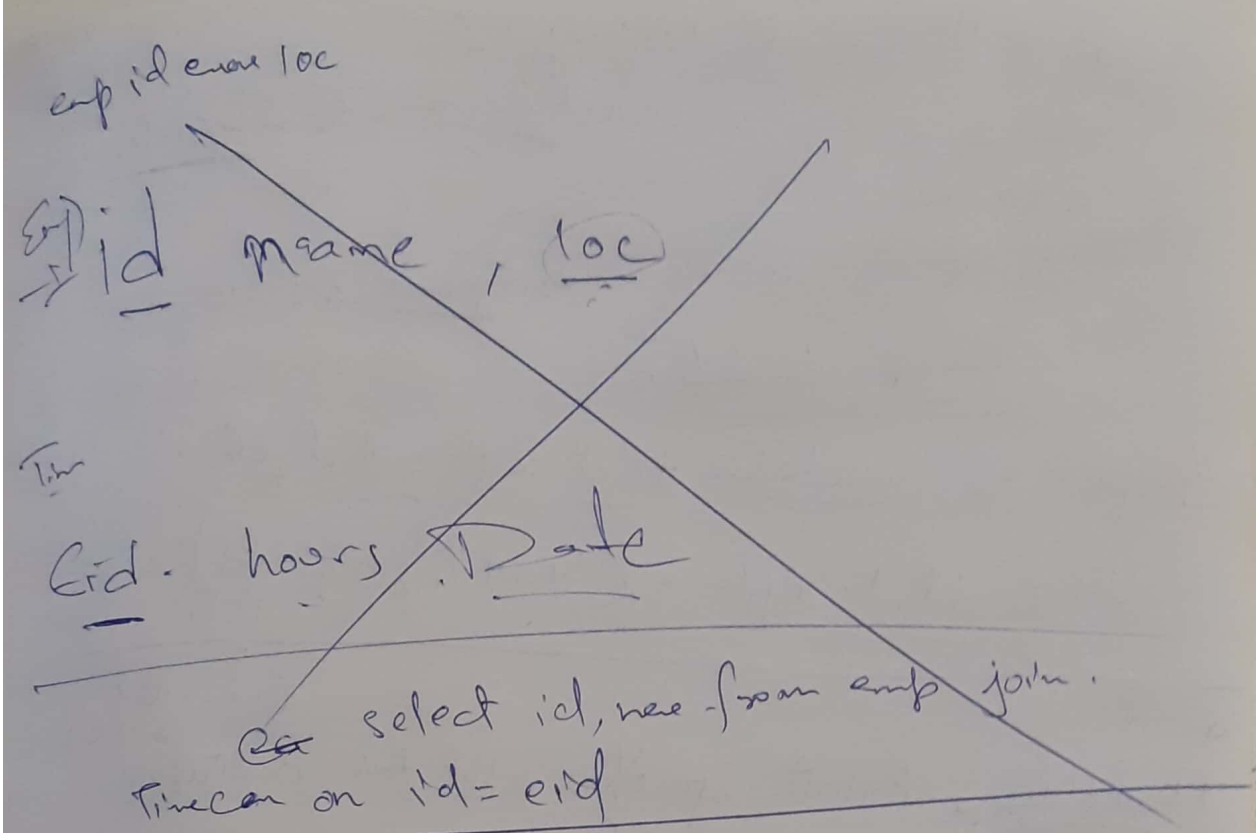
Reading up input data

var UIController = (function() {
return {
getInput: function() {
return {
type: document.querySelector('.add__type').value
description: ''.''.''('.add__description').value
value: '' ('.add__value').value
};
};
})();

returns an object

fn:- var input = UIController.getInput();
console.log(input)

Tip:- Place all event listeners in 1 place



~~Ex~~ select id, name from emp join.
Timecard on id = eid

How to set up a proper DS for budget controller

```
var data = {
    allItems: {
        exp: [],
        inc: []
    },
    totals: {
        exp: 0,
        inc: 0
    }
};
```

→ data.allItems[type].push(newItem)
// It works in JS.

Adding new items to the UI i.e Dom manipulation

```
addListItem: function (obj, type) {
    // create HTML string with placeholder text.
    // Replace the placeholder text with actual data.
    // insert the HTML into Dom
}
```

addListItem: function (obj, type) {
html = '<div class="item clearfix" id="income-%id%">
<div class = " .... "> <

→ Now, we can make the values dynamic, such as:-
"income-%id%"

//Replace the placeholder text with actual data.

newHtml = html.replace('%id%', obj.id).

(see from 3:40)

Placeholder attribute specifies a hint that describes the expected value of a text field

---

• Now, we replace placeholder with actual data, using a replace method.

var newhtml;

→ It searches the string & replaces it with obj.id

newhtml = html.replace('%id', obj.id);

↳ It is a string variable

newHtml = newHtml.replace('%description%', obj.description)

newHtml = newHtml.replace('%value%', obj.value)

// Insert the HTML into DOM.

Syntax:- Adding elements to DOM

element.insertAdjacentHTML(position, text);

→ position is the position relative to element & must be 1 of the following strings:-

a) 'beforebegin' → Before the element itself

b) 'afterbegin' → just inside the element, before its first child.

c) 'beforeend' → " " " , after its last child

d) 'afterend' → After the element itself.

visualization of position name.

```
<!-- before begin -->
<p>
   <!-- after begin -->
   foo
   <!-- before end -->
</p>
<!-- after end -->
```

Ex:-

```
var d1 = document.getElementById('one');
d1.insertAdjacentHTML(
    'afterend', '<div id="two">
        two</div>');
```

(or)

document.querySelector(element).insertAdjacentHtml('beforeend', newHtml)

element → ".myclass"

---

clearing the input fields

c

```
clearFields: function() {
    var fields =
fields = document.querySelectorAll(DomStrings.inputDescription +
        ', ' + DomStrings.inputValue);
```

1) str is a syntax to select 2 (or) more selectors

A list is returned instead of array

→ List doesn't have many methods like array

so we transform it to an array

//so we use call to access all array methods

```
var fi=
Array.prototype.slice.call(fields); // it is a way to trick the
                                    this object  class method to think
                                                 that we passed an
                                                 array, & it should return the array
```

↳ it is a function constructor for all arrays

```
fi.forEach(function(current, index){
    ↳(for each element of array, the fn is applied.
    current.value="";
});
fieldArr[0].focus();
}
```

How to convert field inputs to numbers

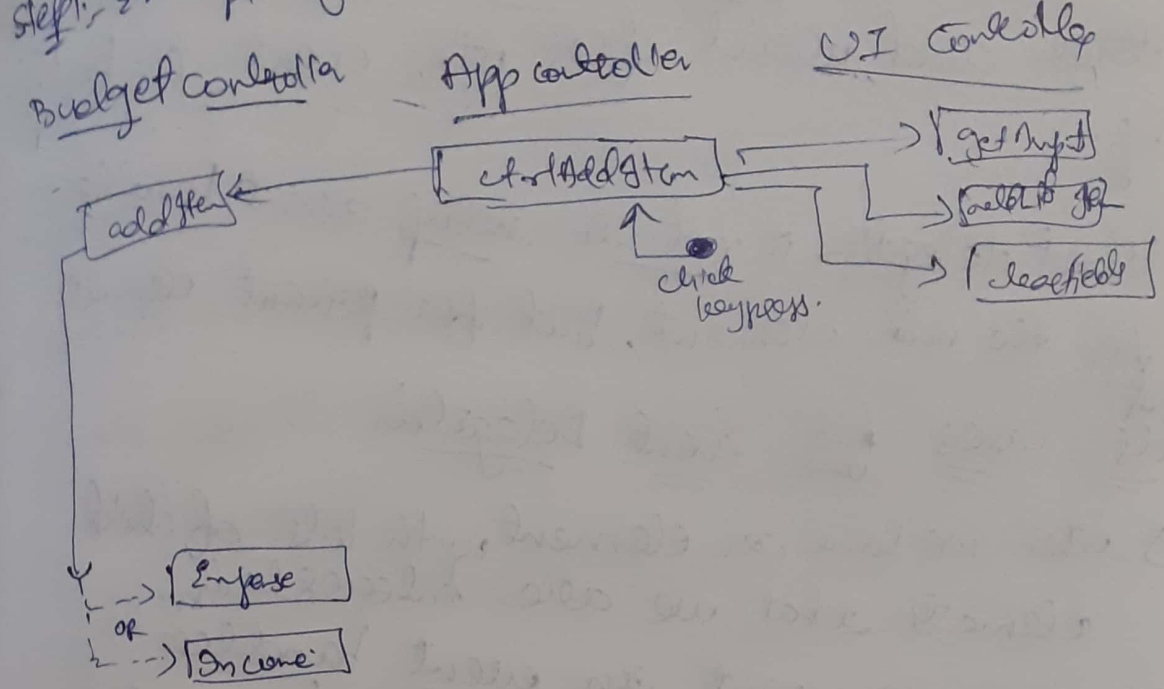
```
if(input.description !== "" && !isNaN(input.value)
        && input.value > 0){
```

3

Only changing the text content:-

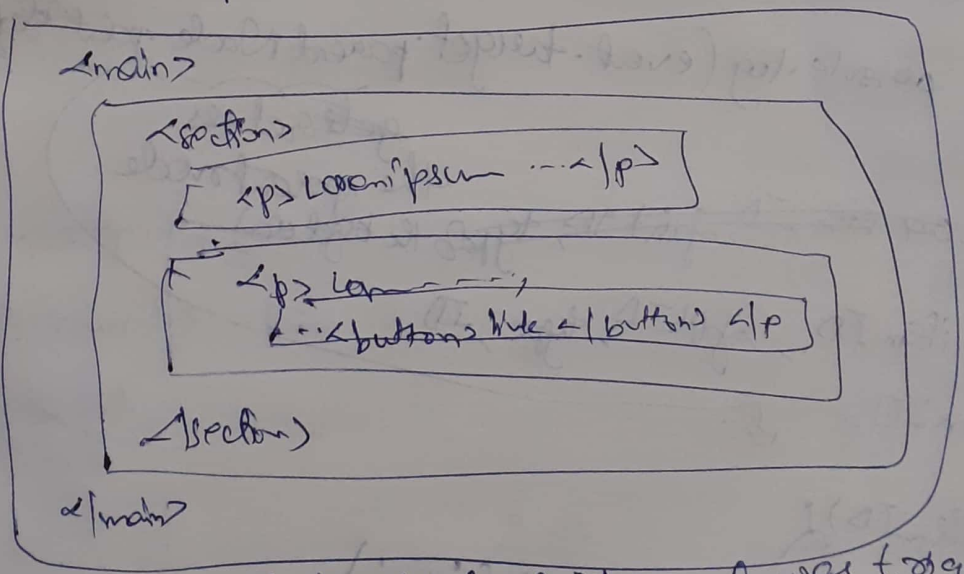
```
document.querySelector(DOMstrings.budgetLabel).
        textContent = obj.budget
```

Step 1:- Example of architecture



Event Delegation

1) Event Bubbling :- when an event is triggered on some DOM element i.e (by clicking a button) then exact same event is triggered for all of the parent elements



2) The element in which first event was triggered is called as Target Event.

3) Event Delegation :- If event bubbles up in DOM tree, & if we know then

we can attach parent element to the DOM tree and wait for the event to bubble up.

→ Event delegation is not to setup event handler for the main element, but for parent element

use cases for Event Delegation

1) when we have an element with lots of child elements that we are interested in.
2) when we want an event handler attached to an element that is not yet in DOM, when our page is loaded.

Example of event delegation (we set up for parent element, in whose child we are interested in)

```
var ctrlDeleteItem = function(event) {
    console.log(event.target.parentNode.parentNode)
```

gets/retues the parentnode of the target obj.

```
var itemID, splitID, type, ID;
itemID =
if (itemID) {
    splitID = itemID.split('-');
    type = splitID[0];
    ID = splitID[1];
};
```

clean above values inputs method.

Deleting an item from UI:

a) Removing an item from UI:

deleteListItem: function (selectorID){

var el = document.getElementById(selectorID);

el.parentNode.removeChild(el);

} // works, used, but this is how it works

/

Use of different string methods to manipulate strings:

formatNumber: function (num, type) {

var numSplit, int, dec;

num = Math.abs(num); // dont need to declare a num var, as the parameter will act as regular variable & will be

num = num.toFixed(2); // method of number prototype, it will round the number to only 2 decimal places

numSplit = num.split('.');

numSplit = num.split('.'); // separates integer & decimal pt.

int = numSplit[0];

if (int.length > 3) {

int = int.substr(0, int.length - 3) + ',' + int.substr(int.length - 3, int.length);

} // 2310 → 2,310
// 23510 → 23,510

dec = numSplit[1];

type === 'exp' ? sign = '-' : sign = '+';

return type + ' ' + int + dec

}

How to get current date by using Date object constructor

```
displayMonth: function () {
    var now, year, month, months
    var now = new Date(); // it will return today date
    year = now.getFullYear();
    month = now.getMonth(); // returns the month number (0-11) 0 is 0 based
    months = ['January', 'February', 'March', ...]
    month = months[month];
    document.querySelector(DOMstrings.dateLabel)
        .textContent = months[month] + ' ' + year;
}
```

How & when to use "change" events

→ document.querySelector(DOM.inputType).addEventListener('change', UICtrl.changedType)

(the name of the event; the name of the method)

→ changedType: function() {

```
var fields = document.querySelectorAll(
    DOMstrings.inputType + ',' + DOMstrings.inputDescription + ',' + DOMstrings.inputValue);
```

... now, we use nodeList for each.

```
nodeList.forEach(fields, function (ele){
    ele.classList.toggle('red-focus');
});
```

Next Generation of JS i.e ES6 & beyond

→ All that we have discuss earlier, still valid & important, all that changes is syntax and ~~support~~ other new updates.

① Variable declaration with let & const.

let is a var, and const is used for constant values.

```
const name = 'Sara'
let age = 18
name = 'Jake' // cannot change & its immutable
age = 21
```

Difference.

② In var the variables are in function-scope. i.e

```
function dl(test){
    if(test){
        var a = 10;
    }
    console.log(a);
}
```

③ In let & const, variables are in block scope.

```
function dl(test){
    if(test){
        let a = 10
    }
    console.log(a); // Reference error
}
```

④ In ES6, to access variables, we can do the following:

```
function dl(test) {
  let fn;
  const yOB = 1990;
  if (test) {
    fn = 'Jake';
  }
  console.log(fn + yOB)
}
```

⑤ Another difference b/w ES5 & ES6 is:—

- In ES5, we were able to use variables, b4 they were declared, & we used to get the console.log as undefined:

```
console.log(fn) // undefined
var fn = 'Jake';
```

- In ES6, we get Reference Error.

```
Reference error in ES6
```

Strings in ES6

① ES5
```
console.log('This is ' + fname + ' ' + lname + ...)
```
Now in ES6, we have template literals using tilde character:-

```
console.log(`This is ${fname} ${lname}.
  Today is ${calcAge(yOB)} years old.`)
```

```
const n = `${free} ${free}`;
console.log(n.startsWith('j'));
console.log(n.endsWith('sm'));
console.log(n.includes('oh'));
console.log(`${free}`.repeat(5))
```

Arrow functions in ES6

```
cost years = [1990, 1965, 1982, 1937];
let ages = years.map(el => 2016 - el);
```

```
ages = years.map((el, index) => `Age
el.ret ${index + 2}: ${2016 - el}y.`);
console.log(ages)
```

→ If we have more lines of code, then we need to use return statement.

```
ages = years.map((el, index) => {
  const now = new Date().getFullYear();
  const age = now - el;
  return `Age el.ret ${index + 2}: ${age}.`
});
```

this keyword

→ Arrow functions do not have this keyword instead they use the this keyword for a function ~~not callable~~ they are written for.

ES 6

```
function Person(name) {
    this.name = name;
}

person.prototype.myFriends =
function(friends) {
    var arr = friends.map(el => `${this.name} is friends with ${el}`);
    console.log(arr);
}
friends = ['michel', 'Jacky', 'Tiger'];
// Create a new object:
new Person('Mike').myFriends(friends)
```

Destructuring

1) Extract data from data structure like obj (or) arr

```
// ES 5
var john = ['John', 26]
var age = john[1]
```

```
// ES 6 aka Destructuring
const [name, age] = ['John', 26];
console.log(name);
console.log(age);
```

```
const obj = {
    fname: 'J',
    lname: 'S',
};
```

→ `const {fname, lname} = obj;`
`console.log(fname);` → (These 2 const variable names should match the keys)
`console.log(lname)`
(or) we can use aliases as: —

`const {fname: a, lname: b} = obj;`

→ we can use destructuring in function calls too

```
function calAge(year) {
    const age = new Date().getFullYear() - year;
    return [age, 65 - age];
}
const [a, b] = calAge('1990');
```

Arrays

```
const boxes = document.querySelectorAll('.box');
```

Array.from() → allows us to create an Array instance from an array-like (or) iterable object.

→ `console.log(Array.from('foo'))` // ["f", "o", "o"]

→ `console.log(Array.from([1, 2, 3], x => x + x));`

// [2, 4, 6]

Spread operator → useful to expand array elements.

1) function addFourAges (a, b, c, d){
return a+b+c+d;
}
const num3 = addFourAges(...ages); //expands all the array elements

2) const f1 = ['A', 'B', 'C'].
const f2 = ['D', 'E', 'F'].
const f3 = [...f1, ...f2] // merging the array elements

Rest Parameters:- allows to pass arbitrary number of parameters to a function.

function isFullAge (...years)
console.log(years);
}
isFullAge (1990, 1999, 1965)

Note: spread operator is used in function call and Rest Parameter operator is used in function declaration.

* To add params along with rest parameter
function isFull(limit, ... years) {
:
}
isFull (16, 1990, 1999, 1997, 2016);

Default Parameters

function Smithy (frN, yOB, lN = 'Smith', nationality = 'amreki') {
this.fn = frN
this.yOB = yOB;
}
var john = new Smithy ('John', 1990);
//we can also over-ride the default args as
var mike = new Smithy ('mike', 1991, 'Jack', 'Sudan');

Maps:-
→ A new DS, its a key-value DS i.e. hash maps.
→ In maps, we can use anything as key i.e number, boolean, string, function, etc, whereas in objects, we have only strings as keys

Ex:-
const question = new Map(); // created a empty map
question.set('question', 'what is latest version?') — keys, value; ↳ add data to map.
question.set(1, 'ES5').
question.set(2, 'ES6').
question.set(3, 'ES2015').
question.set(4, 'ES7');
question.set('correct', 3);
question.set(true, "correct Answer").
question.set(false, 'wrong, try again!').

Quiz game.

→ To retrieve da, we pass key:-

```
console.log(question.get('question'));
```

→ To obtain the size:-

```
console.log(question.size)
```

→ question.delete(4);
key

→ question.has(4) → check whether key is present or not.

→ question.clear() → delete all elements in map

→ Maps are iterable.

```
question.forEach((value, key) =>
    console.log(`This is ${key} & its set to
    ${value}`))
```

```
for (let [key, value] of question.entries())
    console.log(`This is ${key} & value
    is ${value}`);
```

entries() returns all entries of

```
const ans = parseInt(prompt('write the correct answer'))
console.log(question.get(ans === question.get('
    correct')));
```

---

classes (makes it easier than function constructors) (Inheritance)

```
class Person {
  constructor(name, yob, job) {
    this.name = name;
    this.yob = yob;
    this.job = job;
  }
  calculateAge() {
    var age = new Date().
      getFullYear() -
      this.yob;
    console.log(age);
  }
  static greeting() {
    console.log('Hey!');
  }
}

const john1 = new Person
('John', 1990, 'teacher');
Person.greeting();
```

Every class should have constructor.

→ We can also have static methods, i.e they are attached to the class, instead of attributes and are not inherited by class instances

---

→ We can only add method to classes, but not properties

Inheritance:-

```
class Athlete extends Person {
  constructor(name, yob, job, olympicGames, medals) {
    super(name, yob, job); // call to superclass cons
    this.olympicGames = olympicGames;
    this.medals = medals;
  }
}
```

```
wonMedal() {
    this.medals++;
    console.log(this.medals)
}

const johny = new Athlete('Johny', 1990, 'swimming', 10);
johny.wonMedal();
johny.calculateAge();
```

→ Array of objects

```
const allParks = [new Park('green park', 1987, 0.2, 315),
new Park('National', 1899, 2.9, 3541), new Park('Hyde
    1990, 8.4, 999)];

function reportParks(p) {
    p.forEach(el => el.treeDensity())
}

reportParks(allParks);
```

Asynchronous JS → some code that keeps running in background, while other code is still executing.

→ In synchronous JS, all instructions are executed one-after, the other, in the order in which they are written.

→ For asynchronous JS, we use setTimeout func.

```
<script>
const second = () => {
    setTimeout( () => {
        console.log('Async Hey there')
    }, 2000 ) // After 2 seconds have passed the callback func will run.
}
    callback  milliseconds

const first = () => {
    console.log('Hey there');
    second();
    console.log('The end');
}
first()
</script>
```

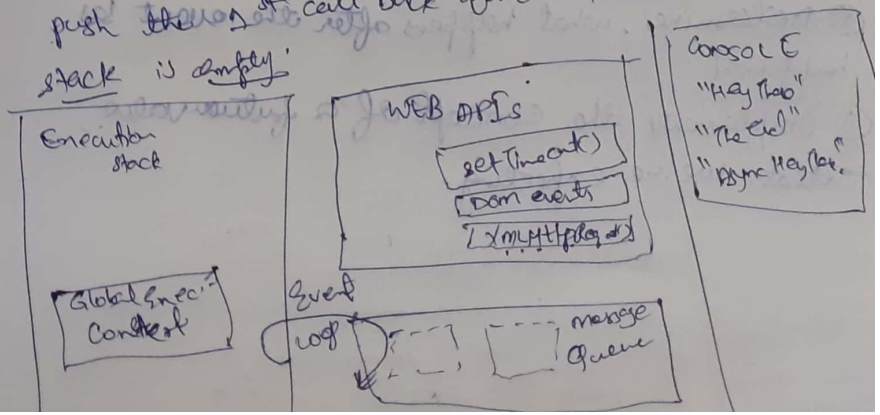
o/p
Hey there
The end
Async Hey there

---

① It allows asynchronous functions to run in the "background".

② We pass the callbacks that run once, the function has finished its work.

③ Event loop :- Its job is to constantly monitor, Message Queue & Execution stack, and its push the callback function, when the stack is empty.



Asynchronous JS with callbacks

```
function getRecipe() {
    setTimeout(() => {
        const recipeID = [523, 883, 432, 974];
        console.log(recipeID);
        setTimeout((id) => {
            const recipe = {title: 'Fresh Tomato Pasta',
            publisher: 'Jones'};
            console.log(`${id}: ${recipe.title}`);
        }, 1000, recipeID[2]);
    }, 1500);
}
getRecipe();
```

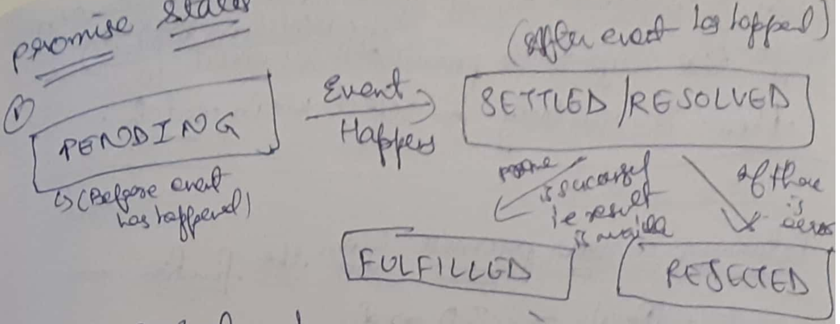
it is passed as parameter.

// After 1.5 second (i) will be executed & then after 1 second (ii) will be executed.

Promises (Hardest Topic in JS)

1. Object that keeps track about whether a certain event has happened already or not.
2. Determines, what happens after the event has happened.
3. Implements the concept of a future value that we're expecting.

promise states



2. We can produce & consume promises, i.e when we produce a promise, we create a new promise and send a result using that promise, when we consume a promise, we use callback functions for fulfillment or rejection.

<script>

```
const getIDs = new Promise((resolve, reject) => {   // executor function
    // If event is successful, we call resolve else reject.
    setTimeout(() => {
        resolve([523, 883, 432, 974]); // Result of the promise is passed via resolve.
    }, 1500);
});
```

i.e after 1.5 seconds, we say promise is successful & we want to return the data.

// Now consuming promise using then().

```
getIDs.then(IDs => {
    // allows us to add event handler, when the promise is successful.
    console.log(IDs);
}); // IDs will receive parameter from resolve.
// then() also returns a promise

getIDs.catch(error => {
    console.log('Error!!'); // resolve ki jagah reject.
}); // catch allows the promise
```

//Iska ye matlab hai ki, ... if ajan mein se objects (datamain success) (resolve) laye aaya to the promise is returned to .then. & similarly in reject mein .catch.

Returning a new promise from the function

```
const getRecipe = recID => {
  return new Promise((resolve, reject) => {
    setTimeout(ID => {
      const recipe = {title: 'Fresh tomato pasta',
                      publisher: 'Jones'};
      resolve(`${ID}: ${recipe.title}`);
    }, 1500, recID);
  });
};
```

// We have produced a promise now how to consume it
we can achieve it by chaining the promises.

```
getIDs
  .then(IDs => {
    console.log(IDs);
    return getRecipe(IDs[2]);
  })          ↳ It will return a promise object
  .then(recipe => {
    console.log(recipe);
  })
```

ES8 ASYNC | AWAIT was introduced in ES8 to make the concept of Promises, easier.

- Async/Await was designed to consume promises, we produce the promises the same way as earlier.

```
async function getRecipesAW() {   → It also returns a promise.
(keyword)
  const IDs = await getIDs; // we obtain the result from first promise i.e getID which we created earlier.
  console.log(IDs);
  const recipe = await getRecipe(IDs[2]); // promise is returned
  console.log(recipe);
  return recipe
}
getRecipesAW() // This function runs in the background.
               ↳ synchronous callback
```

⇒ Also await, can only be used in async function only

Description

1) The await expression causes async function execution to pause until a Promise is settled & to resume the execution of async function after fulfillment.

2) When resumed, the value of await expression is that of fulfilled/or rejected Promise.

3) If the value of expression following the await operator is not a Promise, it is converted to a resolved promise.

// On the previous program, since a promise is returned by async function.

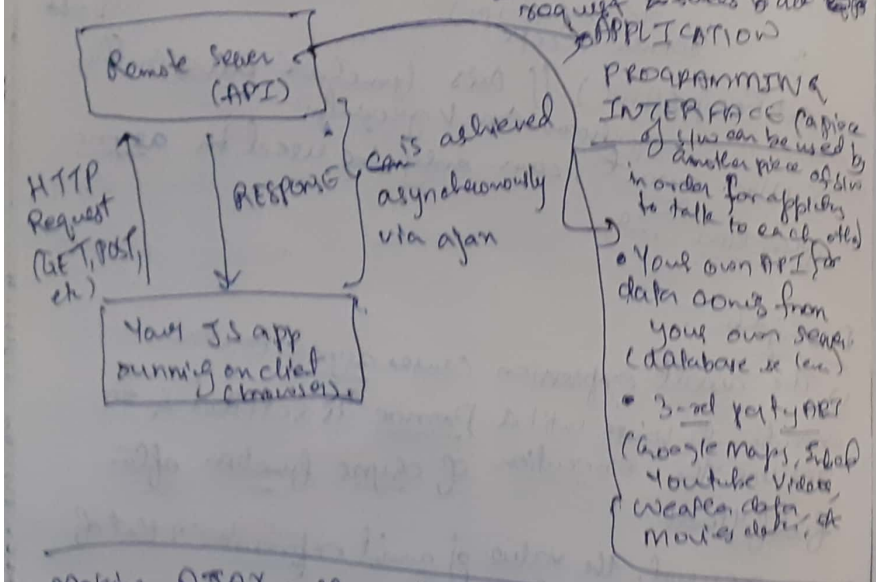
```
getRecipeAux().then(result => console.log('${result} is the best ever!'));
```

(result → consume promise)

AJAX

AJAX → Asynchronous JS & XML.

Allows us to asynchronously communicate with the server.



Making AJAX calls with Fetch & Promises

<script>

```
fetch('https://www.metaweather.com/api/location/search/?query=london')
```

(1: URL to which API is located (2) URL to which we are sending the request) (location/search/?query=london)

Note:- Same-origin policy:- prevents to make request from a domain different than the one that AJAX request

---

So, to find a way out; Cross-Resource Origin Resource sharing was developed.

→ But for metaweather it still doesn't work, so developers proxy (or) channel the AJAX request to own server, and then send the data to ours

→ So we prefix the request with:-

```
https://crossorigin.me/ ...url
```

```
fetch('https://crossorigin.me/....')  // fetch API gets data as a promise (or) consumes a promise
```

(It will produce a promise.)

```
.then(result => {
    console.log(result)
})
.catch(error => console.log(error));
```

→ Response body: Readable stream, is obtained, so we use .json() to convert JSON to JS

```
fetch('.....')
.then(result => {
    return result.json()   // It also returns a promise & is resolved
})
.then(data => {
    console.log(data)
})
.catch(error => console.log(error));
```

Note:- To obtain specific fields, we use:-

```
.then(data => {
    const t = data.consolidated_weather[0]
    console.log(t)
```

→ Like in the previous example, we were using a static value in URL path, so if we want to fetch inside a function, we can achieve dynamic content.

```
function getWeather(woeid) {
  fetch(`https://.....${woeid}`)
}

getWeather(1038917);
getWeather(46710);
```

Note: If we send a false request, we get 404 error.

Babel :- used to transform ES6/ES Next to ES5, so all browsers understand them.

webPack :- separating different parts of project into modules.

model - View - Controller Architecture (MVC)

Now consuming the promise with async / await (It replaces the .then)

```
async function getWeatherAW(woeid){
  const result = await fetch("https://.....");
  const data = await result.json();
  console.log(data);
}
getWeatherAW(2487171);
getWeatherAW(26178);
```

// Handling errors in async / await.

→ we use try / catch.

```
async function getWeatherAW(woeid){
  try{
    const result = await fetch(...);
  }
  catch(error){
    console.log(error);
  }
}
```