Module 2 – 38

Module 3 – 20

Module 4 – 28

Module 5 – 60

Modue 6 – 20

Module 7 – 42

Module 8 – 124

Module 2 : Javascript

uses

- lightweight, adds interactivity to HTML (& responsive), executes code without pre-lim compilation, embedded directly into HTML pages

- Compatable across platforms and devices (simplification)
- can be combined with HTML and CSS
- Supports server side code than can be written cross-platform runtime engines using Node.js
- implements MEAN architecture (MongoDB, Express.js, Angular.js, Node.js)
- creates cookies
- implements client side scripts

- Javascript with HTML Attributes :

(i) <script> tag for defining client side code.
(ii) may contain scripting statements or points to an external file through "src" attribute.
(iii) It returns a collection of particular node's attribute as an object, NameNodeMap.
(iv) The program can access these nodes through index number starting from 0.

- Javascript HTML DOM elements :

(i) using id method : var ele = document.getElementById("first")
(ii) using tag name : var x = document.getElementByTagName("p")
(iii) using class name : var x = document.getElementByClassName("—") → returns list
(iv) using CSS selectors
(v) using HTML object collections

- Javascript with CSS :

```
<!DOCTYPE html>
<html> <body>
<h1 id = "id1" > myProgram says Hello ! <br> And click
below ! </h1>
<button type = "button" onclick = " document.getElementById (`id1`).
style.color = " red " >
click me ! </button>
</body> </html>
```

- Adding JS :

```
<script language = " javascript " type = " text/javascript " > ----
</script>
```

- where to put in scripts :

(i) b/w <head> tags of an HTML doc

(ii) with <body> tags of an HTML doc's body

(iii) In external file or anywhere within webpage linking script from its HTML doc

- JS operators :

(i) Arithmetic operators : +, -, /

(ii) Assignment operators : =, +=, %=

(iii) Comparison operator : ==, !=, >=

(iv) Logical operators : &&, ||, !

(v) Type operators : typeof, instanceof

```
var a = 5 ; var b = 12 ; var c = a+b ;
document.getElementById("myProgram").innerHTML = c ;
```

- x = 5 ⟶ x is assigned the value "5"

  x == k ⟶ checks if values of x & k are equal or not

  x === k ⟶ " " " & datatype of variables is = or not

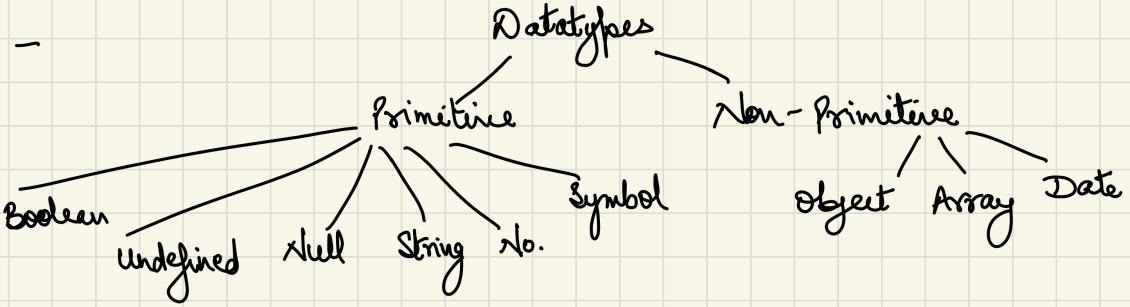
- Data types :

```
var num = 5                                   // number
var lastname = "Peterson"                     // String
var x = { firstname : "Gary", lastname : "Haskel" }; // object
```

If no & a string are used in one statement, JS will always treat or consider the number as a string.

- Data Types in JS are dynamic. Same variable can be assigned different values.

-



- typeof operator :

typeof "" returns string
typeof 314 returns number
typeof "Garry" returns string

- Primitive data : have no additional properties or methods

(i) Boolean : returns true & false
all values are true until defined otherwise
pre-defined false values in JS are "", 0, NaN, undefined, null.

(ii) Undefined : when a variable is not assigned a value, it returns "undefined" as its value and type

(iii) Null : data type of null (value) shall be of the type "object" (datatype)

(iv) Strings : var str = "Hello" // double quotation
var str1 = 'hello' // single quotes

All primitive data types are immutable, in JS. Strings are mutable in C/C++.

↳ can't be modified but another string can be created based on original string operation

String inside string ?

- Strings Method : Concatenating strings

① var str = "Ju" + "Javascript"

② var str1 = "Hello"
var str2 = "Jim"
var str3 = str1.concat(" ", str2)
document.getElementById("id").innerHTML = str3

- String Method : finding a string within a string

① indexOf() — returns the position of first occurence of a particular text (can take two parameter) → starts from 0

② lastIndexOf() — returns position of last index of a text
↳ returns -1 if text is not found

- search() Method :

① Also returns the position of first occurence of a text to be found

② Doesn't take second param. for specifying the start position for search

- Primitive Data Type – Numbers (int, float, hexadecimal, octal, exponential)

var int = 1000

var float = 100.5

Special numeric values = Infinity, NaN

```
alert (1/0) ;  // Infinity
```

```
alert ("Text String"/2);  //NaN, such division is erroneous
```

result of an incorrect mathematical operation that's conventionally not been defined

- Number Methods – Returning No.s

① toString() — returns numerical values as a string

② toExponential()

```
var x = 7.456
x.toExponential(2)  // 7.46e+0
x.toExponential(4)  // 7.4560e+0
```

③ toFixed()

```
var x = 13.656
x.toFixed(2)    // returns 13.66
```

④ toPrecision()

```
var x = 13.656
x.toPrecision(2);    // returns 14
x.toPrecision(4);    // return 13.66
```

⑤ toValue() —— returns a number from numerical

①-⑤ return datatype "string"

- Non primitive datatype :

① Object — collection of related data or property defined by "key" and "value" written as "key : value" pairs

```
var person = { name: "Gary", age: 45 }    // object person defined
```

An object is a reference data type. Variables are assigned a reference value which point to the location in the memory of code where object is stored.

② Array — var array1 = [val1, val2 ... , valn]

```
<html> <body>
<h2> JS Arrays </h2>
<p id = "myProgram" > </p>
<script>
var car = ["Bentley", "Ferrari", "Mercedes"]
document.getElementById("myProgram").innerHTML = cars;
</script>
</body> </html>
```

③ Date – by default an object-type data

JS display date and time as a full-text using

Wed Jan 15 2019 17:02:06 GMT+0530 (India Standard Time)

① newDate() — displays current date and time in default view

② newDate(year, month, ...) — create a new date object with whatever no. of parameters represent

7 No.s – year month day hour min sec millisec

6 No.s – year month day hour min sec

1

2 No.s – year month

But

③ new Date(milliseconds) : var xd = new Date(78912)

④ new Date(dateString) : creates a new date object from date specified in the form of a string

– var xd = new Date(0) // returns "Thu Jan 01, 1970 00:00:00 UTC"

– JS functions :

- functions are building blocks of a code which runs the program and executes the assigned task to it.

- why functions :

① To reuse the code multiple times after defining it once

② Get varied results in different forms when used in diff. threads or arguments

```
function myfunc (p1, p2) {
    return p1 * p2
}
```

- using functions as variable values :

```
var text = "The temp is" + toFahren(75) + " F".
```

- using variables as local / global :

① Local variables : - variables declared from within JS f^n
- can be accessed only from within f^n

eg:
```
function myfunc () {
    var a = 7 ;
    return 2a ; }
```

② Global variables : - variables accessed by f^n which when defined outside of it

eg.
```
var = 7 ;
function myfunc () {
    return 2a ;
}
```

- Local variable

① recognizable by their fn only, same name can be used in diff. fn

② Their scope is similar to that of a fn in a code.

③ dies when fn ends

Global Variable

lifetime till time the program lives

Module 4 : Node.js :

- History of Node.js :

① Google's V8 Javascript engine was embedded and an event loop into a C++ program and called that program Node

② Introduction of a package manager for Node.js called npm, making it easier for dev. to share source code of Node.js lib.

③ Implementation of native ver. of windows Node.js

④ First ever Windows supporting node.js was released

⑤ Neutral Node.js foundation was formed.

- What is Node.js : Runtime env. for developing server side app.

features : ① light-weight, fast
② JS based
③ free
④ open source
⑤ cross platform
⑥ Node.js uses JS on server side

- Where to use Node.js :

① Data Intensive Real Time App.
② Data Stream App.
③ Single Page App.
④ I/O bound App.
⑤ JSON API based App.

- Where Not to use :

① App. involving heavy comp. & CPU need is ↑
② App. involving MySQL, Postgres, Oracle

- Why Node.js :

① built on chrome's JS runtime for building fast & scalable n/w app
② uses an event driven, non-blocking I/O model → DIRT

② uses JS —— no other lang. is req.

④ Large ecosystem for open source lib

⑤ efficient for prototyping & agile dev.

Node.js is not a lang. or framework. it is a runtime env. for executing JS code.

- Architecture of Node.js :

① Node.js contains JS engine to execute JS code outside of the server.

~~document.getElementById('');~~

fs.readFile( ) — reading file

http.createServer( ) — requesting API server

Node has chrome V8 JS engine along with some add' modules that are not available while working inside browsers.

- features — VLENS

① single-threaded but scalable

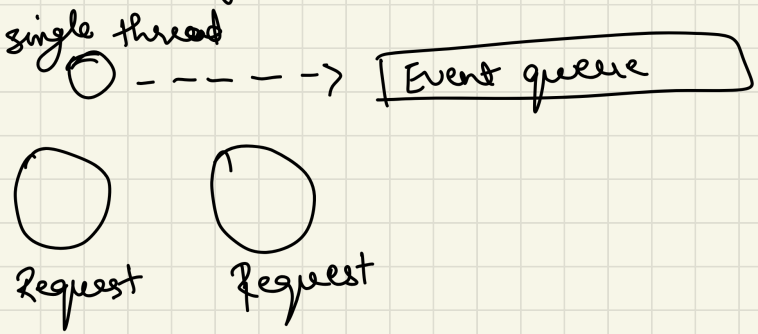
② event driven & Asynchronous

③ License

④ No buffering

⑤ very fast

- working of Node.js :



Following are the working of Node.js

- I/O is Asynchronous and Event Driven
- A single thread is used to handle multiple requests
- Avoids buffering of data
- Every API of Node.js library also supports asynchronous functioning; and hence are non-blocking in nature.
  - This allows Node.js-based-server to work without waiting for an API to return data.
- These applications simply output the data in chunks

- ***Event Queue*** receives the messages.
- Whenever result is to be returned, the message is conveyed to ***Event Queue***.
- Node continuously manages and monitors ***Event Queue***.
- The Node takes out the event from the ***Queue*** and process it based on the required results.

- REPL Env. :

⊞ + R → cmd → node → Read - Eval - Print - Loop

REPL commands :

.help : Display help on all the commands

tab Keys : Display list of all commands

.save filename : Save current Node REPL session to a file

.load filename : Load specified file in current Node REPL session

ctrl c : terminate the current command

" (twice) : exit from REPL

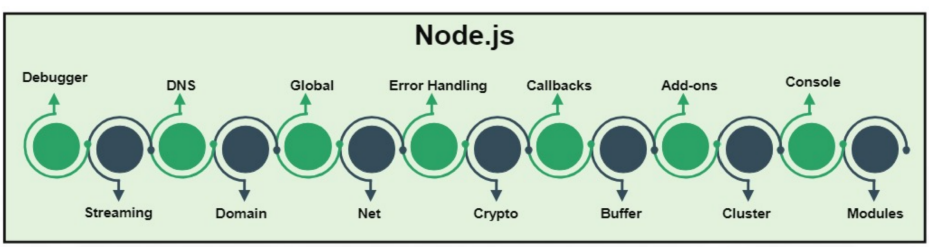
" d : "

.break : exit from multiline exp.

.clear : "

- use console.log(__) to save variables

Components of Node.js



- Modules: global scope, local scope, file containing JS code = module, will have explicitly expost the f" & variable name in another module

To view the properties of a “module”, we can console log the “module” onto our node.js command prompt.

App.js

```
1
2   console.log(module);
3
```

```
C:\Users\Guru\mydigitalnode>node app.js
Module {
  id: '.',
  exports: {},
  parent: null,
  filename: 'C:\\Users\\Guru\\mydigitalnode\\app.js',
  loaded: false,
  children: [],
  paths:
   [ 'C:\\Users\\Guru\\mydigitalnode\\node_modules',
     'C:\\Users\\Guru\\node_modules',
     'C:\\Users\\node_modules',
     'C:\\node_modules' ] }

C:\Users\Guru\mydigitalnode>
```

Local Modules: (var. & fn defined locally)

Module Export: Export object

module — current module

exports —

An anonymous function, can be attached to exports object and exported to any other module to return a result. This is similar to object.

For example, an addition function to be imported in another module to receive values and return the added sum.

moduleOne.js

```
1
2   const add = (num1, num2) => {
3       return num1 + num2;
4
5   };
6
7   module.exports = add;
8
```

moduleTwo.js

```
1
2   const add = require ('./moduleOne');
3
4   console.log(add(11,23));
5
```

eg

For example, consic … ; exported wrapped using insid

moduleOne.js

```
1
2   module.exports = {
3       firstName: 'ABCXYZ',
4       lastName: 'LTD'
5   }
```

```
1
2   var company = require('./moduleOne.js');
3   console.log("Name of the Company is",
4   company.firstName + company.lastName);
```

```
Name of the Company is ABCXYZLTD.
```

NOTE: Editable code in the **'note'** section

eg 2

An anonymous function, can be attached to exports object and exported to any other module to return a result. This is similar to object.

For example, an addition function to be imported in another module to receive values and return the added sum.

moduleOne.js

```
const add = (num1, num2) => {
    return num1 + num2;

};

module.exports = add;
```

moduleTwo.js

```
const add = require ('./moduleOne');

console.log(add(11,23));
```

Module Exports : Export Class :

- JS can treat its fn like a class. Any fn acting like class can be exported

Example:

myXmodule.js

```
module.exports = function (company, location)
{
this.company = company;
this.location = location;
this.result = function () {
return ("Welcome to" + ' ' + this.company + '\n' +
    "We are based out of" + ' ' + this.location);
}
}
```

App.js

```
var app = require('./myXmodule.js');
var myXdigital = new app('ABCXYZLtd.', 'Delhi');
console.log(myXdigital.result());
```

```
Welcome to 'ABCXYZLtd.'
We are based out of Delhi
```

- Module export : Loading Module from Separate Folder :

var xprogram = require ('./Container/Xmodule.js');

- Operating System : Module to get info. about current os.

os module has methods like :

① freemem() : returning available free memory in bytes as an integer

② userInfo() : returning info. about current user

③ uptime() : returning uptime of machine

```
const os = require ('os');

var totalMemory = os.totalmem();

var freeMemory = os.freemem();

console.log(`Total Memory is ${totalMemory}`);

console.l
```

- HTTP Module

Example: Creating a web server using createServer() method of Server class

The object,

`server = http.createServer();` is an event emitter

We can now use `server.on()`, `server.emit()`, or `server.listen()`

Calling ***server.listen*** to create a port and listening on that port.

Every time there's a request sent, the ***server*** will create an Event

The on() method is used to handle the event

```
const http = require('http');

const server = http.createServer();

server.on('connection', (socket) => {
  console.log('New connection...');
});

server.listen(3000);

console.log('Listening on port 3000...');
```

like building request.

Building backend app can be developed using HTTP module.

This module has various classes with different methods : Class.http.Agents()

Class.http.ClientRequest()

const http = require('http')

Example: Creating a web server using createServer() method of Server class

The object,

`server = http.createServer();` is an event emitter

We can now use `server.on()`, `server.emit()`, or `server.listen()`

Calling ***server.listen*** to create a port and listening on that port.

Every time there's a request sent, the ***server*** will create an Event

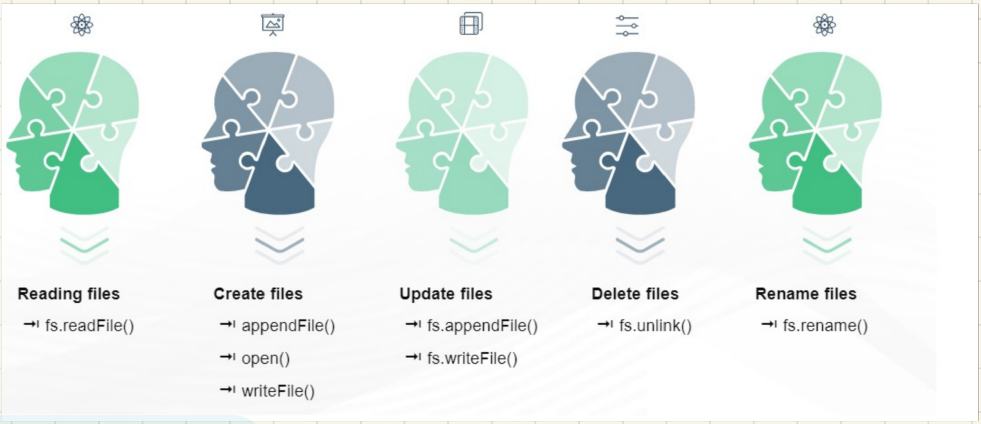
The on() method is used to handle the event

```
const http = require('http');

const server = http.createServer();

server.on('connection', (socket) => {
  console.log('New connection...');
});

server.listen(3000);

console.log('Listening on port 3000...');
```

- File Systems: Module offers an API to interact with file system in a way, which is closely modeled around POSIX fn.

The completion callback is always taken as last argument by asynchronous forms.

The first argument is always reserved for throwing an exception, if the operation was completed successfully then first argument will be null or undefined.



Example: Reading from a text (txt) file synchronously.

Input File: myXinput.txt

myXinput.txt

```
1   Welcome to Course
2
3   A place where you'll learn all about coding, programming language, and
4   designing databases from 'front' to 'back' literally!
```

Main File: ***App.js***

App.js > ...

```
1
2   var fs = require('fs');
3
4   var data = fs.readFileSync('myXinput.txt');
5   console.log(`Synchronous read: \n${data.toString()}`);
6   console.log("Program Ended");
```

Output

```
Synchronous read:
Welcome to DigitalE1

A place where you'll learn all about coding, programming language, and
designing databases from 'front' to 'back' literally!
Program Ended
```

NOTE: Editable code in the **'note'** section

Example: ***Writing*** into myXinput.txt and replacing the content inside

Main File: ***App.js***

```
var fs = require("fs");

var data = `Welcome to the new DigitalE1.
It's Bigger and Better`;

fs.writeFile("myXinput.txt", data, (err) => {
  if (err) console.log(err);
  console.log("Successfully Written to File.");
});
```

myXinput.txt

myXinput.txt

```
1   Welcome to the new DigitalE1.
2   It's Bigger and Better
```

Output

```
C:\Users\Guru\mydigitalnode>node App.js
Successfully Written to File.
```

NOTE: Editable code in the **'note'** section

Flags: indicating the behaviour of the file to be opened

Flag	Description
r	Opening file for reading. An exception would occur if the file does not exist.
r+	Opening file for reading and writing. An exception would occur if the file does not exist.
rs	Opening file for reading in synchronous mode.
w	Opening file for writing. The file would be created (if not existing) or truncated (if existing).
wx	Similar to 'w' except it would fail if path exists.
w+	Opening file for reading and writing. The file would be created (if not existing) or truncated (if existing).
wx+	Similar to 'w+' except it would fail if path exists.
a	Opening file for appending. The file would be created if it does not exist.
ax	Similar to 'a' except it would fail if path exists.
a+	Opening file for reading and appending. The file would be created if it does not exist.

eg:

Example: Creating a File for writing into it

`fs.open('./digitalInput, w+, function(err,fd){`

```
App.js > ...
2   var fs = require("fs");
3
4   console.log("Going to open file!");
5   fs.open('digitalInput.txt', 'w+', function(err, fd) {
6       if (err) {
7           return console.error(err);
8       }
9     console.log("File opened successfully and ready to be written!");
10  });
```

List of files before

- App.js
- moduleOne.js
- moduleTwo.js
- myXinput.txt
- myXmodule.js

New file, **digitalInput** created

- App.js
- digitalInput.txt
- moduleOne.js
- moduleTwo.js
- myXinput.txt
- myXmodule.js

```
Going to open file!
File opened successfully and ready to be written!
```

NOTE: Editable code in the **'note'** section

The fs module could also be used to reading ***html*** files on server.

Syntax code:

```
const http = require('http');
var fs = require("fs");

http.createServer(function (res) {
    fs.readFile('test.html', function(data) {
      res.writeHead(200, {'Content-Type': 'text/html'});
```

The status 200 represents OK.

Module 5 : React.js :

- ES6 Module is an external file that contains reusable code that can imported into other JS files can contain variables, classes, fn's and more

usage of let & const :

The let statement: enables the programmer to re-define a same variable within the scope which is limited to a certain block of code

```
<script>
var  num = 11;          // Here, num is 11
{  let num = 22;        // In this block of code, num values to 22
}                       // Out of the above block, num is again 11
```

The ***const*** statement: It assigns a fixed value to something.

```
<script>
var  num = 11;          //  Here, num is 11
{  const num = 22;      //  In this block of code, num values to 22
  var num = 33;         //Error
}                       //  Out of the above block, num is again 11
```

ES6 arrow f" :

Arrow f"s do not contain their own 'this'. ∴, can't be used for defining or calling object method.

Arrow functions are syntactic alternative to standard function expression in a compact manner

```
// ES5 let courses = ['CSS', 'HTML', 'JavaScript',
'React', 'Redux'];

        //ES5
        courses.map(function(module){
        return module.length;
        });
        ▶ (5) [3, 4, 10, 5, 5]
```

```
// ES6 let courses = ['CSS', 'HTML', 'JavaScript',
'React', 'Redux'];

        //ES6
        courses.map((module) => {
        return module.length
        });
        ▶ (5) [3, 4, 10, 5, 5]
```

```
let courses = ['CSS', 'HTML', 'JavaScript',
'React', 'Redux'];
//ES 6 one Liner
courses.map(module => module.length);
▶ (5) [3, 4, 10, 5, 5]
```

NOTE: Editable code in the **'note'** section

Using ⇒ in Promises :

Arrow functions are syntactic alternative to standard function expression in a compact manner

The syntax below:

```
return new Promise(function (resolve, reject){
setTimeout(function(){
```

Can be written as:

```
return new Promise((resolve, reject)=>{
setTimeout(()=>{
```

ES6 – New Built-In Method :

Array.find()

To find and return the value of the first array element which passes the test condition.

```
[9, 99, 121, 11, 111,].find(x => x > 100)
121
```

Array.findIndex()

To find and return the ***index value*** ([1], [3], etc.)of the first array element which passes the test condition.

```
[9, 99, 121, 11, 111,].findIndex(x => x > 100)
2
```

NOTE: Editable code in the **'note'** section

Alternate method :

```
var firstval = Array_name.find(myXfunction);
function myXfunction(value, index, array){
    return value > 100;
}
```

Number Type checking :

isFinite( ) : to find & return value of first array element which

isNaN( ) : " " " " index " " " "

Classes :

properties are defined in constructor( )

eg

Example: Demonstrating using ***method with parameter*** in the ***class*** and accessing its properties

```
class Course {
constructor (crtype) {
this.crtype = crtype;
}
Inst(name) {
console.log(`'${name}' designs ${this.crtype} course`);}};
let myCourse1 = new Course("JavaScript");
myCourse1.Inst("DigitalE1");
let myCourse2 = new Course("React and Redux");
myCourse2.Inst("DigitalE1");
```

```
'DigitalE1' designs JavaScript course
'DigitalE1' designs React and Redux course
```

NOTE: Editable code in the '**note**' section

Inheritance using extends:

```
class Square extends Polygon {
  constructor (x, length) {
  super (x, length)
  this. length = length
  }}
```

Default value for parameters:

ES6 allows the programmer to declare a variable have that variable used in a function with the same default value throughout the program without using the variable again.

```
<html>
<h3> Default Parameter Values </h3>
<p id="Xba"></p>
<script>
function myXfunction(val1, val2 = 11) {
// num2 is 11 if not passed or undefined

return val1 + val2;
}
document.getElementById("Xba").innerHTML =
"val1 + val2 = " + myXfunction(22);
</script>

</html>
```

Default Parameter Values

val1 + val2 = 33

NOTE: Editable code in the '**note**' section

spread operator :

ES6 allows the elements of an array or string to be spread into the form of either literal elements or parameters of individual function or both.

```
var elements= [ "Hi", true, 7 ]
var arr=["Digital", "E1", ...elements]
```

```
▸(5) ["Digital", "E1", "Hi", true, 7]
```

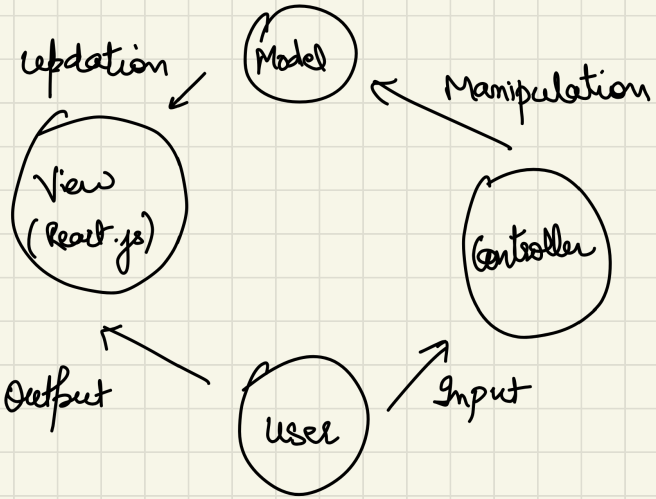
```
var str = "DigitalE1"
var chars = [ ...str ]
```

```
▸(9) ["D", "i", "g", "i", "t", "a", "l", "E", "1"]
```

exponential : $3^{**}4 = 3^4$

iterators :

Intro To React:



React is a frontend lib for JS.

features of react:

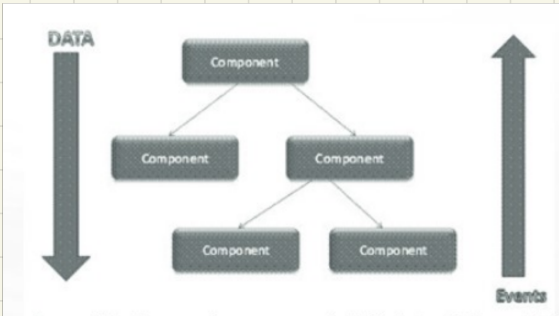
1. Declarative design → dynamic design, rapid rerendering, easy debug
2. Component Based lib → component, out of DOM, internal state – TodList
3. Unidirectional data flow → components are immutable, callbacks. Data is synced automatically

what makes react frontent :

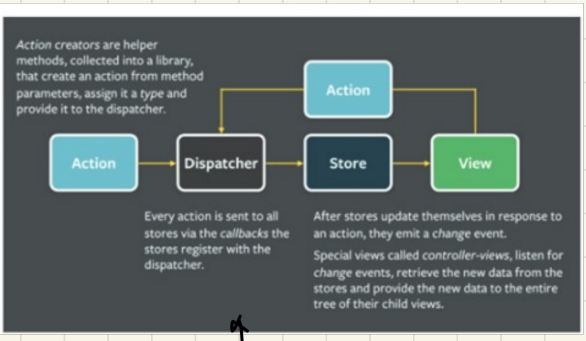
Frontend Library

- Being a "View" Layer, React application does not require to know where the data is being input from.
- It does not "listen" to the server updates. User feeds data into the application and renders the interface.
- REACT's sole responsibility is to update the interface with the new data
- It's the User's responsibility to write logic for states, map the props to UI and insert event handlers updating/changing the states and props

React can be executed on server site as well as client side. But even if view is generated on the server site. it doesn't mean that application is "listening" to the server.



Source: https://www.c-sharpcomer.com/article/what-and-why-reactjs/



unidirectein data flow → flux architecture, explain this

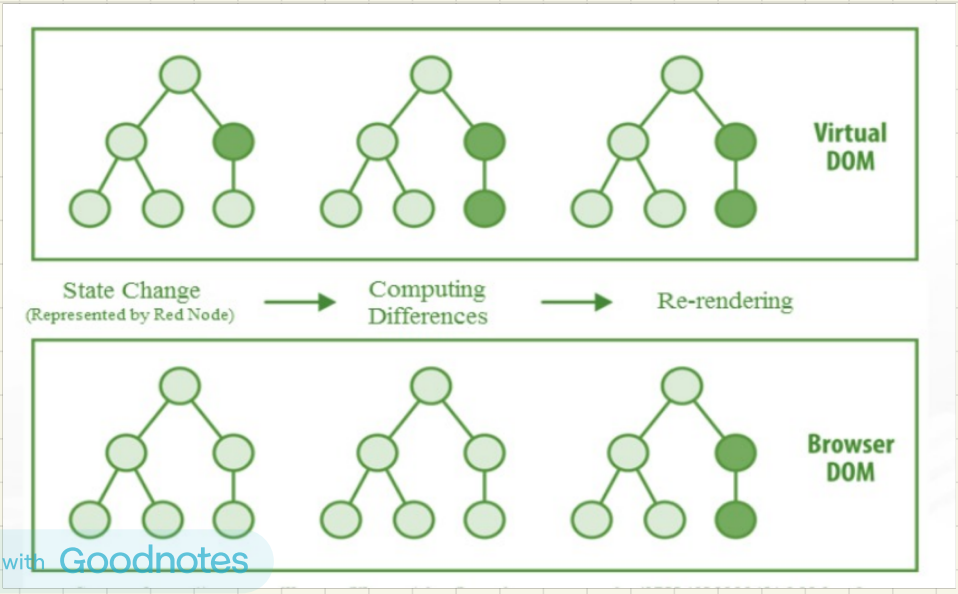
why we need react : (23 slide)

1. Dom is slow
2. React uses virtual dom
3. Simple & easy to learn
4. Reusability

5. Robust re-rendering with virtual DOM
6. Native libraries
7. Unidirectional Data binding
8. Easy Testing

How does React work:

1. REAL DOM → user interface of application
2. Why DOM gets slow → as app grows, tree-str.
3. Virtual DOM is faster bcoz → batch updates
   - i. virtual tree
   - ii. comparison
   - iii. - batch update



④ Leveraging virtual DOM → observable pattern, the dev. doesn't need to know what is happening behind the scenes & only state has to be updated

⑤ render() function
↳ for updating UI
↳ returns new tree → using setState() method

React updates virtual dom & then updates only corresponding in real dom.

npm — 26

Setting up React:

- Installation of Node.js also includes installation of Package Manager.

The npm stands for the Node Package Manager that manages the installation of different packages and modules for the Node.js platform. It helps the node platform to find the installed module and use them accordingly. Further, it manages dependency conflicts very efficiently and neatly

① Install node.js
② open node.js command prompt from start menu
③ npm install -g create-react-app
④ create-react-app <project_name>

<project_name>
├ react
├ react-dom
├ react-scripts

⑤ npm start

- JSX : Syntactical extension of JS syntax, used to create React elements. Renders these elements to React DOM.

var ele = <h1> Hello </h1>

why JSX : ① faster than JS
② optimization → JS
③ Better optimization → React, highlight errors
④ easy to handle UI with JS

- Embedding JS exp in JSX :

```
const fname = 'Participant';
const jsxelement = <h1>Hello, {fname}</h1>;
ReactDOM.render( jsxelement,
document.getElementById('root')
);
```

Hello, Participant

– Nested Element in JSX :

More than one element can be used and wrapped inside one container element. Here, we showcase **div** containing **three** nested elements, <h1>, <h2> and <p> inside it.

```
import React, { Component } from 'react';
class App extends Component{
   render(){
      return(
         <div>
          <h1>  Course  </h1>
          <h2>Hello, Participants</h2>
          <p>    Course  designs best tutorials on JS and React.</p>
         </div>
      );
   }
}
export default App;
```

Course

Hello, Participants

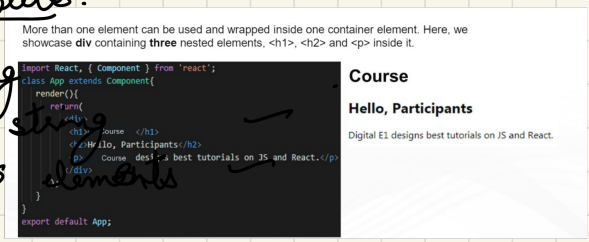
Digital E1 designs best tutorials on JS and React.

– JSX Attributes :

① Camel casing

② " " for string

③ {} for JS elements



– JSX comments :

{ /* */ }

– JSX styling and representation as Object :

The content of an element can be styled as well as represented as Object as follows:

```
import React, { Component } from 'react';
class App extends Component{
   render(){
     var myXStyle = {
         fontSize: 40,
         fontFamily: 'Times New Roman',
         color: "blue"
     }
     return (
         <div>
            <h2 style = {myXStyle}>Hello, DigitalE1</h2>
         </div>
     );
   }
}
export default App;
```

Hello, DigitalE1

NOTE: Editable code in the **'note'** section

- State of Component:

Entity of the component that can change over time in response to action of the user or system event taking place.A state of the component contains information and certain data about the component in a given time which determines the component's behavior and how it is going to render. The state also determines dynamicity and interactivity of the component and represents its local state at a given point of time.A state must be kept very simple. The state at any given point could be updated using setState() methodThe user interface will trigger the changes and updates when setState() is called upon. The state can only be accessed, modified, or updated from inside the component or directly by the component. Before the time any interaction between components occurs, the initial state can be set using getInitialState()For instance, the developer needs to create a container element like to keep state of all the components which requires information from the state.

- Defining State:

initial state has to be defined for a component by adding a constructor using this.state, inside render().

Example: Creating state of the component

```
import React from 'react';
class App extends React.Component {
 constructor() {
    super();
    this.state = { displayComp: true };
    }
    render() {
      const comp = this.state.displayComp ? (
        <div>
        <p><h3>Setting up and Creating the State of the Component</h3></p>
        </div>
        ) : null;
        return (
          <div>
          <h2>     Course  Welcomes everyone to React! </h2>
          { comp }
          </div>
          );
      }
}
export default App;
```

Course Welcomes everyone to React!

Setting up and Creating the State of the Component

- changing the State:

Defining, changing and updating the state of the component using setState() method

Button is added to the render () method; clicking on the button will call the displayComp() method showing the desired output.Participants should run the code by themselves and see the result.

```
import React from 'react';
class App extends React.Component
{ constructor() {
super();
this.state = {displayComp: false}; }
console.log('Component this', this)
;this.toggleDisplayComp = this.toggleDisplayComp.bind(this);}
toggleDisplayComp(){ this.setState ({displayComp: ! this.state.displayComp});}
render() {
return( <div>
<h2> DigitalE1 Welcomes everyone to React!</h2>
{ this.state.displayComp ? ( <div> <p> <h3>Defining, Changing and Updating
State of the Component </h3> </p> <button onClick = {this.toggleDisplayComp}>
Show Less </button> </div>): ( <div> <button onClick =
{this.toggleDisplayComp}> Read More </button> </div> ) } </div>)}
}export default App;
```

- Props of Component:

Props indicate Properties and are ***read-only*** entities of the component.

Storing the attributes of a value, and hence do not change ***(immutable).***

Enables communication between different components by letting data to pass through from one component to another.

Can be passed to the component in the same manner as arguments inside a function.

The ***this.props*** is used to make Props available to the component.

Add ***props*** to ***reactDom.render()*** method in the ***main.js*** file of the project to create and use immutable data.

Code for Props : (Uppercase for custom)

Example: Using Props as argument inside a function

```
function Greetings(props) {
  return <h1>Hello, {props.userName}</h1>;
}

const element = <Greetings
userName=" Participant ." />;
ReactDOM.render(element,
document.getElementById('root'));
```

Hello, Participant

NOTE: Editable code in the **'note'** section

It is not imperative to always insert props inside the reactDom.render() element. You can also set **default** props directly on the component constructor.

App.js

```
import React, { Component } from 'react';
class App extends React.Component {
    render() {
        return (
            <div>
              <h2> Welcome to { this.props.name } </h2>
              <p> <h4> Using Props </h4> </p>
            </div>
        );
    }
}
App.defaultProps = {
  name:    "Course"
}
export default App;
```

Main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.js';

ReactDOM.render(<App/>,
document.getElementById('app'));
```

Welcome to Course

Using Props

NOTE: Editable code in the **'note'** section

Example: Using Props as immutable object

App.js

```
import React, { Component } from 'react';
class App extends React.Component {
   render() {
      return (
         <div>
           <h2> Welcome to { this.props.name } </h2>
           <p> <h4> Using Props </h4> </p>
         </div>
      );
   }
}
export default App;
```

Main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.js';

ReactDOM.render(<App name = "Course!!" />,
document.getElementById('app'));
```

Welcome to Course

Using Props

NOTE: Editable code in the 'note' section

- Prop Validation :

Prop validation is a process involving a set of tools which helps developers to avoid bugs in future, ensuring correct application of components and making code more comprehensive.

PropType is a set validators which help dev. to catch bugs through data types' validation.

Code ?

- State Vs Props :

State	Prop
entity of component that can change overtime in response to action of user	read only components storing the attribute of a value and hence do not change
contains certain info. about the component which determine component's beha how it is goin	can be passed to the component in the same manner as a function

	Props	State
1.	Read-only and immutable	Asynchronously changeable and mutable
2.	Passing data from one component to another as an argument.	State contains information regarding the component at given time.
3.	Props can be accessed by the child component.	State cannot be accessed by child components.
4.	Communicate between components.	Renders dynamic changes with the component.
5.	Stateless component can have Props.	Stateless components cannot have State.
6.	Enable components to be reused.	Do not allow components to be reused.
7.	External entity controlled by any element rendering the component.	Internal Entity controlled by the React Component itself.

Similarities Between State & Props

Following are the similarities between *state* & *props*:

- Both of them add interactive dynamics to React and its components
- Both act as simple JavaScript Objects
- They can be assigned default values
- When used by the ***this***, both are used in the form of *read-only*

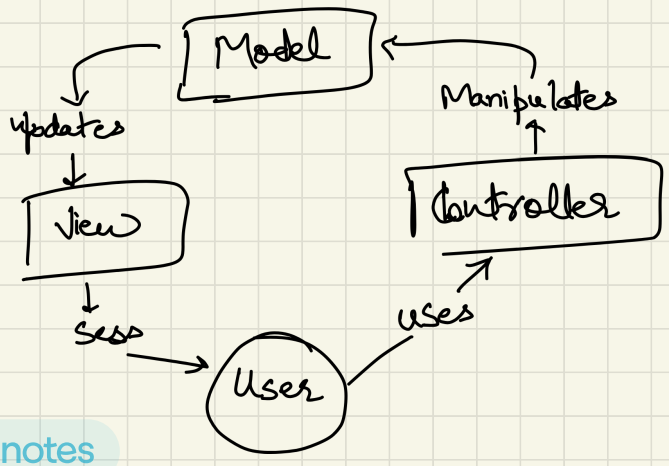
Module 3 : History of frontend :

- Early beginning :

1. Focus on website design
2. Embedding code written within static HTML elements
   ↳ Resulted in static pages

(static page)

- Model - View - Controller :



Model : Central Component of pattern receiving user's input from the controller

View : Responsible for presentation of the model in a particular format

Controller : first line of interface with user accepting its input and converting them into commands for the model or View to work on.

Single Responsibility Principle

Model

Being a central component of the pattern, it is the dynamic data structure of the application, independent of the user interface. Receiving the input from the controller, the **Model** directly manages the logic, data, and rules of the application.

View

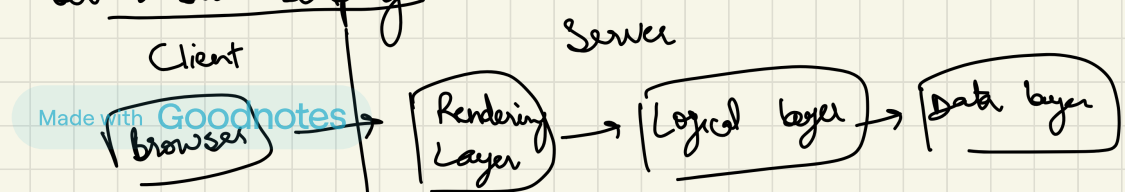
The view represents presentation of the model in a specific format, for example, diagram, chart, or table.

Same information can be presented in multiple formats such as tabular view for accountants, and bar chart for management.

Controller

Acting as a middle man between the user and model, it is the first line of interface through.

- Server-side scripting:



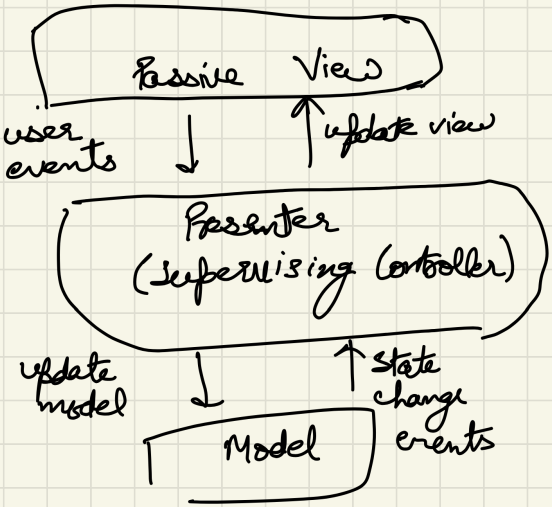
- MVC Architecture :

MVP (Model View Presenter)

MVA → adapter

MVVM — View Model

- MVP Architecture :

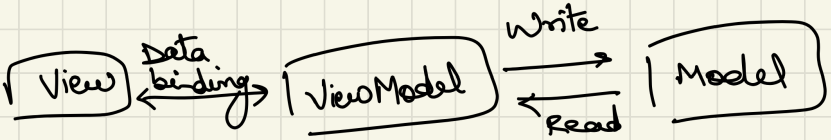


displays data and routes user commands to presenter

retrieves data from repositories (the model) and formats it for display in the view

represents data to be displayed

- MVVM :



why mvvm ? →

1. data binding f^n in WPF
2. removing virtually GUI code from view
3. Making use of framework markup lang (XAML) and create data bindings to view model

separation of roles → focus on UX → higher productivity
than business
logic

Model : refers either to a data access layer or a domain model (an OOP approach)

View : representing the state, appearance and layout of what is displayed on the screen

ViewModel : abstraction of view exposing commands and public prop.

Diff. b/w MVP and ViewModel :

Presenter has reference to view whereas ViewModel doesn't. Instead a view directly binds to properties on the ViewModel to send and/or receive updates requiring a binding technology or boilerplate code generation to do binding.

Development of AJAX :

- Asynchronous Javascript and XML/JSON
- AJAX interacts with API built by Backend

AJAX enables the application to:

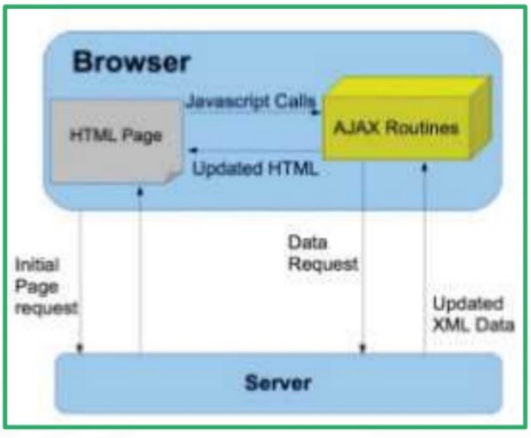
- read data from web server after time page is loaded
- updating web page without reloading
- transporting data to web server in the background

Game Changer: AJAX

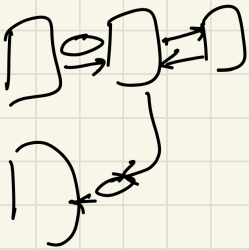
Microsoft Outlook Web Access

Gmail

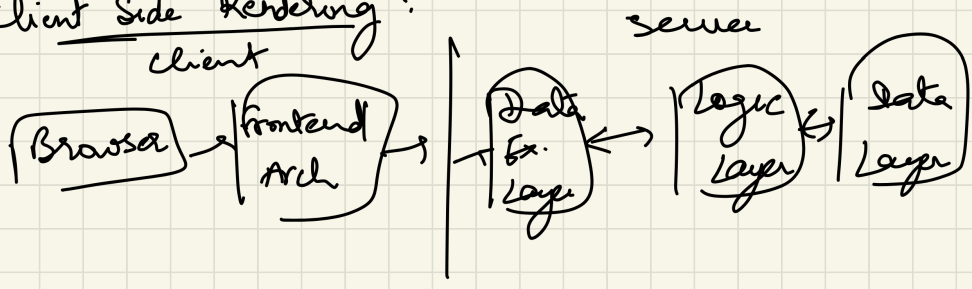
Google Maps



Explanation of AJAX



- Client Side Rendering:



Client-side rendering (CSR) refers to rendering pages using JavaScript and/or DOM directly in the browser. All of the logic, routing, templating and data fetching are handled on the client side rather than the server.

This is also the rationale behind developing **SOFEA:** *that we should stop downloading from the server side and send this responsibility to the client side.*

- With a client-side rendering solution, the request is redirected to a single HTML file and the server delivers it without any content (or
- with a loading screen) until all the JavaScript is fetched and browser compiles everything before rendering the content.
- Under a good and reliable internet connection, it's pretty fast and works well.

The JavaScript engines to the browser and AJAX toolkit have had great development in recent years. The client-side frameworks had a great evolution as well. SOFEA allows for simplicity on the client-side, a more heterogeneous team, and the continued growth of mobile clients.

About DOM: str. of a document, helps dev. to modify

→ Use the following code for adding any arbitrary <section> element at the bottom of your existing script

```
const sect = document.querySelector('section');
```

→ Now, another paragraph can be created using `document.createElement()`

```
const myXcreation = document.createElement('p');
myXcreation.textContent = 'Manav Rachna presents History of Frontend Architecture and Web APIs.';
```

→ This new paragraph can be appended at the end of the section using

```
Node.appendChild():
sect.appendChild(myXcreation);
```

Reactive prog:

- Finally a text node to the paragraph can be added by creating the text node using :
  `document.createTextNode():`
- Now, another paragraph can be created using `document.createElement()`

```
const myXcontent = document.createTextNode(' Arbitrary content to be added.');
```

- And then creating a reference to the paragraph as the link is embedded inside, and appending the text node to it:

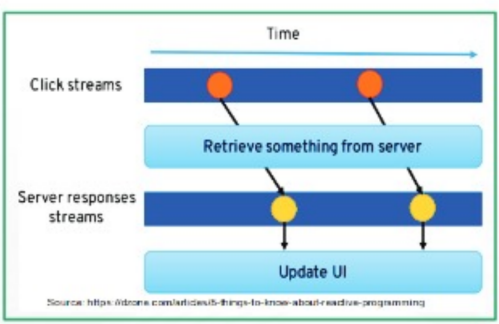
```
const linkPara = document.querySelector('p');
linkPara.appendChild(myXcontent);
```

Reactive Programming

What is Reactive Programming

Programming with asynchronous data streams.

- Raises the level of abstraction of the code in order to create interdependencies between related events that define the business logic.
- Makes the code more concise and enables a developer to avoid having to constantly fiddle with a large amount of implementation details.
- Reactive Programming makes data operate as a "stream" of real-time information flowing, ***to which a program observe and "react" accordingly when a value is emitted***



Reactive Programming makes data operate as a "stream" of real-time information flowing, ***to which a program observe and "react" accordingly when a value is emitted***

While writing a code for Reactive Programming, you will be able to create data streams of everything including click events, hovering event, HTTP requests, ingested messages, availability notifications, and cache events etc.

Streams can be presented anywhere and are cheap, i.e., anything can be developed a stream: variables, user inputs, properties, caches, data structures, etc.

For example, imagine your Twitter feed would be a data stream in the same fashion that click events are. ***You can listen to that stream and "REACT" accordingly.***