Refresher Sessions

Process and Tools

Tools
IDE – VS Code, Eclipse,
Java .java .class .jar test - project organisation Maven / Gradle

Link
https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html
https://dlcdn.apache.org/maven/maven-3/3.8.4/binaries/apache-maven-3.8.4-bin.zip
pom.xml

NodeJS Project
npm init -y

React
npx create-react-app myapp
- package.json where we specify all our project dependecies and commands
- npm install <package_name>
npm start – start the dev server
npm build – to create production build (minified build) in build folder
(nginx – webserver) – Digital Ocean , GCP, AWS
npm test

Process
GIT (SVN (centralised) or CVS). Prefer GIT because it is a distributed VCS
git clone
ComponentA.js (local commit , also called as Staging)
NOTE: Before developing the code for the new feature, ALWAYS clarify the requirements. Ask for mock ups if required.
Unit Test case

Exec UT case
NTH: Cov report

Pull Request (PR)
Address comments or suggestions from the reviewer. If all good then you get the merge approval.
Merge the changes to the central repo

Build Tool – Jenkins
Build Job (scheduled or on-demand)
Deploy the build created on QA Server
Run Functional Test cases.
If the FT cases all pass, then we tag the build as good build.

Logging and Debugging
In Java :
System.out.println
Logger

```
Logger logger = Logger.getLogger(MyClass.class.getName());
```

Logger.log(Level.SEVERE, "Memory too low");

In JS
alert("…"); // Not allowed in production code. Remove them.
Console.log(); // Preferred way of logging. But avoid in prod code.

Debugging Java/JS
React – Redux tool

Unit Testing
What is the unit?
Smallest piece of code that can testing independently. Like a component, or class, function,.

Functional Testing – test the app as a whole. 100TC

JS – Jest + Enzyme
ComponentA.test.js
TC1
TC2
TC3

TC4
TC5

Shallow Component Tests
Mount Component (to test components along with their children also rendered)
Snapshot Testing
Mock Component Testing
Function testing.

import Component from `../components/ComponentB`
ComponentA {

render(){
<ComponentB />
}

}

Here we are creating mock component for `ComponentB, which is NOT yet developed but we know what is the expected output when it is developed.`

```
jest.mock("../components/ComponentB”){
return {
    <div> Helloworld </div>
}

}
```

Agile Methodology
TDD Test Driven Development
BDD Business Driven Development
Understand the requirements
Decide which components to code (System Design)
Write Test cases that describe that use cases (Stories)

Login using UID/password
Upon logging in, the user is taken to his Account Summary page

Login Component
Account Component

Don't code yet, Instead write the Tcs.

```
Account acct = new Account();
acct.setName
..
acct.login();
```

NOTE : For Functional Testing, teams use Selenium.

Best Practices

SOLID principles
Single Responsibility – User class.
There should one and only resaon to change

```
User {
Str name


}

BankSystem{
boolean login(User usr){
….
```

}
}

UserUtil {

saveUser(User usr){
…
}

}

LISKOV SUBS PRINCIPLE

A
B extends A

B b1= new B();

Following should work without any unexpected behaviour

A b1= new B();

Dependency Injection

B b1= <some mechanism that will create B object for us and return>; // Spring FW

INHERITANCE Vs COMPOSTION

Let's say we have an existing class, class A.
Let's we want create another class, class B that should have extra functionality than A.

INHERITANCE Approach : B extends A {…….extra stuff …… }
COMPOSTION Approach:
class B {
A a1…..;

….extra stuff …
}

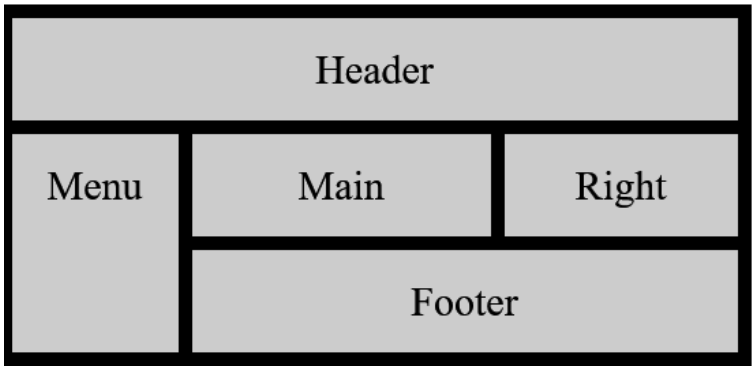
B b1 = new B();
b1.extraStuff ( );
b1. a1. OldStuff();

HTML/CSS
HTML5
– introduced many new semantic control like email, url phone, ...
- nav , section, article, (all for SEO)
- new multimedia controls video, audio,
- canvas
- geo location

CSS – inline style, classes, external stylesheet
CSS3
- CSS3 selector

** **
<p>
<a>

document.queryAll("div p a");
- layouts (Grid layout)
Example : https://www.w3schools.com/css/tryit.asp?filename=trycss_grid_layout_named



src\html
js\…
css
images\...

Java

Basics
Compiled – Bytecode (.class)
Intrepted (JVM)

Java is known be secure programming? What makes it secure? Intrepeted line-by-line. Each line is verified by JVM so that it executes only that code that is safe and permitted. In effect all programs execute in a Sandbox.

Primitives
byte, short, int, long
char, float, double
boolean.

```
short s1 = 170;
int i1 = s1;

int i2=1000000000000000;
short s2 = (short) i2; //Typecasting

float f1 = i2;

char c1 = (char) i2;

boolean b1 = (boolean) i2; //X
```

Q: Is String a datatype?
String is an object. It is NOT a primitive datatype!

In fact every object (like car1 below) is of some datatype.
Car car1 = new Car(); // Here car1 of the type(datatype) Car.

Q1.
class Car {
String name:..

boolean equals( ){
}

String hashCode(){
return
}

}

Q: What do you understand by garbage collection?

OOPs
Q: What is a class?
Class is a blueprint for creating objects.
It is a kind of template for creating objects.

Pillars of OOPS
1. Abstraction and Encapsulation
2. Inheritance – code reuse
3. Polymorphism – polymorphism thru inheritance

Exception Handling
try{

opening a file for reading
...
.....
....

}
catch(ArthmecticException ex){
….
}
catch(FileNotFoundException ex){
..
}
finally{
…..//
file closing.
}

……
…….

NOTE: arrange the catch blocks such that parentclass EXCEPTION type is below the subclass exception type

Collections
int age[] = new int[10];
Student[] students = new Student[10];
students[0] = new Student();

Collections – are flexible
List, Set, Map

List studentList = new ArrayList<Student>();
// search, NOT adding / remove use case
Student student1 = new Student();
studentList.add(student1);

studentList.remove(1);

Teacher teacher1 = new Teacher();
studentList.(teacher1) ;

List studentList = new LinkedList<Student>();
// more objects, adding / remove use case

Set setofStudent = new HashSet<Student>();

Map studentAgeMapping = new HashMap<Student, Integer>();

studentAgeMapping.put(student1 , 510);

studentAgeMapping.get(student1) ;// 510

Collections.sort(studentList);
OR
Collections.sort(studentList, yourOwnComparator);

More Questions:
Q: What is the diff b/w overidding and overloading
Q: What if the diff b/w instance variable and static variables.
Q: what are static imports.
Q: What is the use of final keyword in java?

JDBC
Step1; Know the connection URL
URL = "jdbc:mysql://197.90.12.1:3300//<dbname>"

```
Conection  conn = ConnectionManager.getConnection(URL);
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM
STUDENT);
while(rs.next()){
String str = rs.get("name");
};

 stmt.execute( );
```

Small Problem or Challenges

JavaScript

Before ES6

<html>

<script>

var a1 = new Car();

var x = [“pradeep” , 23, true] ; // we can store all type of datatype values inside the same array.

```
y=”Cat”;

// Functions can be defined like this.
// We don’t have to tell the args of of of certain type.
// We don’t even tell whether we return something or not

function addTwoNumber( p, q ) {
        return parseInt(p) + parseInt(q);
};
```

var result1 = addTwoNumber(2,4);//6

var result2 = addTwoNumber(“2”,4);//6

```
function Cat(n,a){
    this.name=n;
    this.age=a;

    this.getName= function getName(){
        return this.name;
    }
}

var myCat= new Cat("good cat",100 );

console.log("Welcome "+myCat);
console.log("Age "+myCat.age);
console.log("Name "+myCat.getName());
</script>

<script>
//Q: what are diff ways to accept input on your page?
var ageInput = Window.prompt(“….”);

Q: Equality comparison in JS. Always use ===

Q: c!=undefined
```

</script>

Arrays and Objects in JS:

```
var myArr = [];
myArr[0] = 1;
myArr["doorNo"] = 181;
myArr[1] = 78;

console.log(myArr);

var myObj = { };
myObj.age = 75
console.log(myObj);
console.log(myObj.age);
console.log(myObj["age"]);
```

<noscript>
User, looks like you have not enabled JS in your browser.
So, videos, will play on this website
</noscript>

<html>

New in ES6 ES2015
var, let, const
String templates.
Map, Set
New Array methods
Spread and Rest operators
Object destructuring
Arrow Functions
EventHandling
Promises
Modules

var, let, const
let
– works at block scope. once out of the block, they don't exist.

- *syntax wise redeclaring at the same level is not allowed*

```
String templates.
Instead of using  + to contact variable and constantsand literals, use backticks `${x} ….`
```

```
New Array methods
.filter ( x => some boolean condition)

.find  ( x => some boolean condition) = will return the first match if the condition is true. Otherwise undefined
```

```
>> let myArray = [10,20,32,45]
```

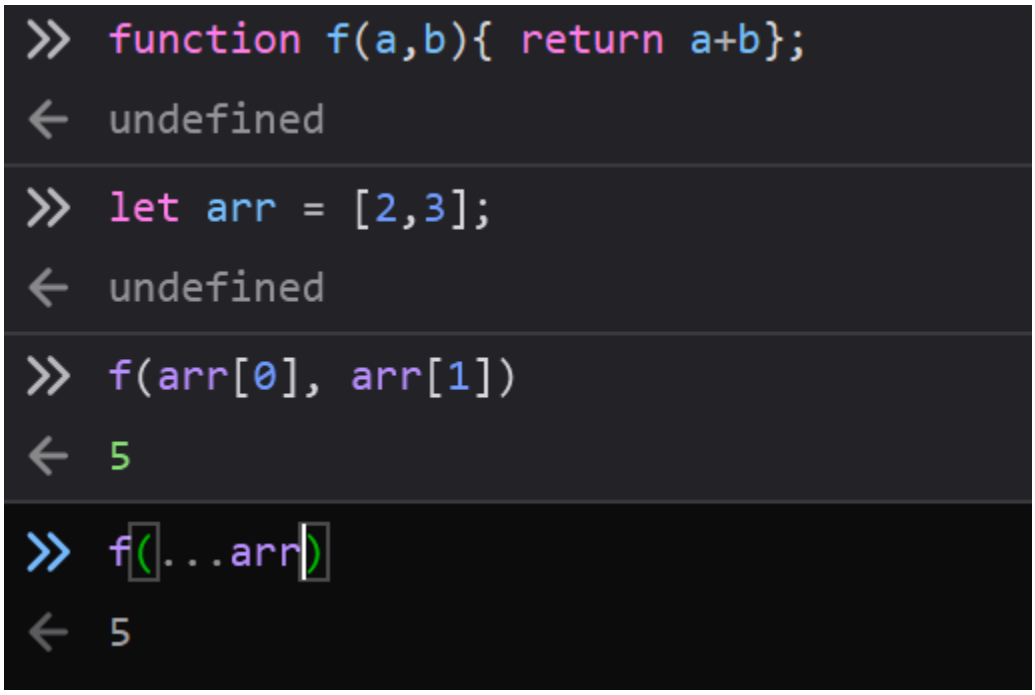
```
myArray.find(x => x>=30)
32
```

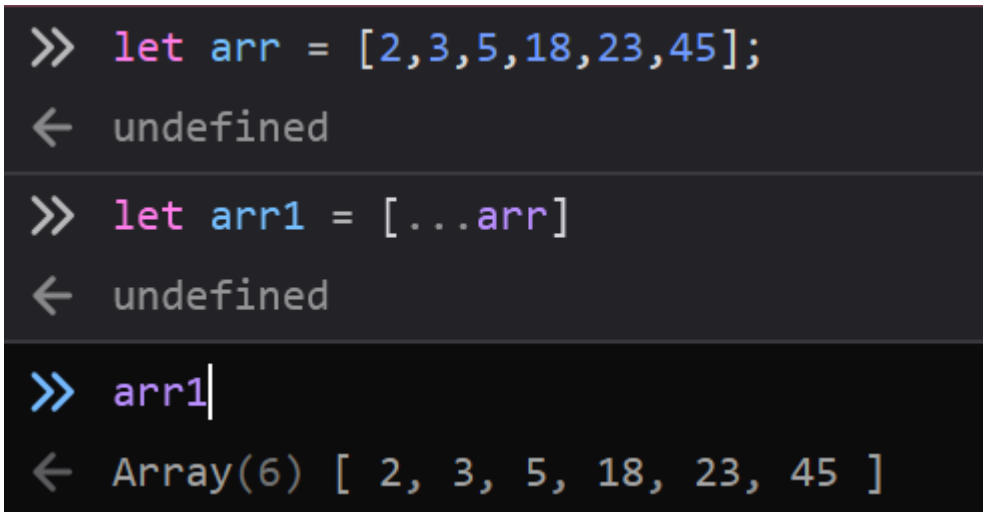
.includes(some value)

```
>> let myArray = [10,20,32,45]
<- undefined
>> myArray.includes(11)
<- false
```

```
>> let myArray = [10,20,32,45]
<- undefined
>> myArray.includes(32)
<- true
```

Spread and Rest operators

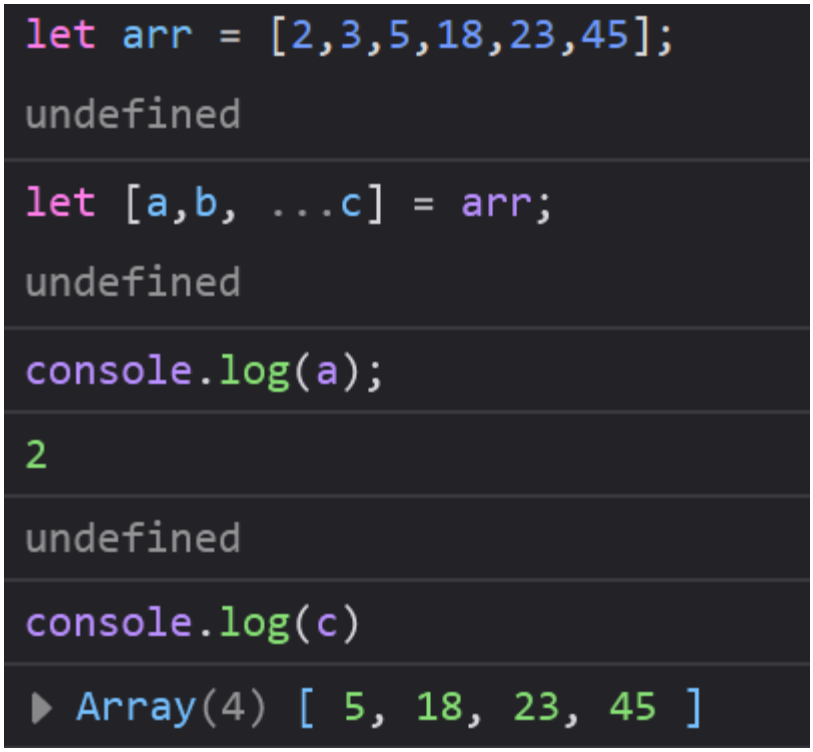


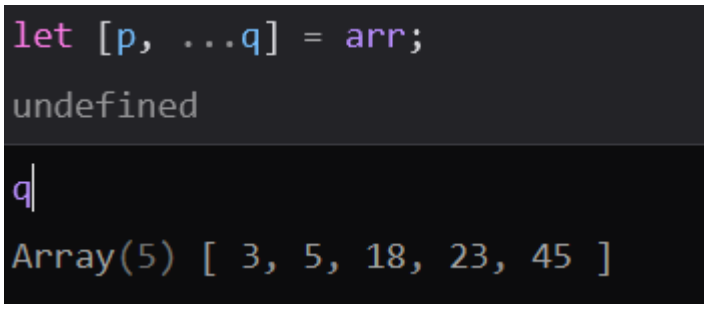


‘

Rest Operator

let arr = [2,3,5,18,23,45];





Arrow Functions
- simplified way of creating function.
Example:
Refer: html_css_js\ES6-arrow-functions.html
- { } can be *omitted for single statement body.*

- *you don't have tell "return" a+b; return can be omitted if it is a single line body.*

EventHandling

We can add/attach/remove eventhandlers for any HTML Element (p , a, div, img, button,…) programmatically inside JS code.
Refer: JS-DOM-Events-2.html

Promises
Objects that do some job and either they will return the result immediately or in future.

```
function resolve(){
////
}

function error(){

}

//Creating promise object.
Var myPromise =  new Promise( function ( resolve, error)
{
console.log(“Hello World”);
if (success) resolve();
else
error();
}
);
```

```
 myPromise.then( “this part will execute when the
promise is  successfully resolved “ );


//Succe3ssful promise
var  promise = new Promise( function(resolve, reject) {
 console.log("In promise. I have completed my job!");
  resolve(100);
});

promise
.then(function(result) {
      console.log("Here is my result " + result );
})
.catch(error => console.log(" in error "));;

Promises are objects that return results in Future, if they
are successful. Otherwise they return error.

/UnSuccessful promise
var promise = new Promise( function(resolve, reject) {
 throw new Error("error happended. I can’t fulfull
promise" );
  resolve();
});

promise.then(function(result) {
  console.log("Promise callback (.then)");
```

}).catch(error => console.log(" in error "));;

TRY CATCH BLOCK

```
function f(a){

        return function g(){
        console.log(a);
        };
    }


    var x= f(2); // x is a function
    x(); // prints 2

    try{
    console.log("inside try block 1"); // CORRECT
     throw Error("Some error happened");
    console.log("inside try block 2");
    }
    catch{
    console.log("inside catch block");
    }

    var y= f(3); // y is a function
    y();// print 3
2
inside try block 1
inside catch block
3
undefined
```

Modules

- solves the name collusion problem.

- author of JS file has complete control of what he wants to share (via export)
- modules work only when served from webserver.

Issue1: Name collusion

```
<html>
<script src=a.js> - x=100
<script src=b.js> - y=10000

<script>
var x=200;
f(x);
console.log(
</script>
</html>
```

Name collusion resolved using import export mechanism

```
<html>
<script src=a.js> - x=100 , f1, export f2, f3
<script src=b.js> - y=10000, g1, g2,g3

<script>
var x=200;
f(x);
console.log(
</script>
</html>
```

```
import A from “a.js”
A.x; // Error, x is not exported, so can’t access x of a.js
A.f1; // Error, f1 is not exported, so can’t invoke f1 of a.js
A.f2; // all good

import XYZ from “b.js”
XYZ.x
```

Modules in JS allows us to let code (data and functions) of one JS to be used in other.
It allows to restrict which data and functions of your file(module) are available to other JS files.
The ones that we want to allow/permit, we export keyword for them.
Others will use import to use the exported data/function of the module they want

REACTJS

Basics
UI Framework (MVC) that provides us a neat way of devlopment reusable components. Using these reusable components, we can built the entire webapp.

React creates VirtualDOM (There is already DOM for every page that is rendered by the browser) from the content we code in render method (JSX stuff)

for I 1to 100000000{this.setState{age:i};

render(){
{this.age}
}

Components can be written in two ways in React
Class based components – render method mandatory
Functional component – return has the HTML to display
PureComponents – are those always render same content. They are class componet that extend PureComponent.

```
class Apple ...{

        render() {
        // before returning we have any number JS statements.
        return JSX code here
        }

}

const arr1=[ ….] ;
```

var x=100
var y={color:”red”, weight:150 }
<Apple name=”myname” age={x} bag={y} />

Props and State:
Props are read-only
Every class component has a built-in variable called state. We access it inside constructor, render or anywhere, saying this.state.
State can hold any object. Like a single value. An array, or an object.

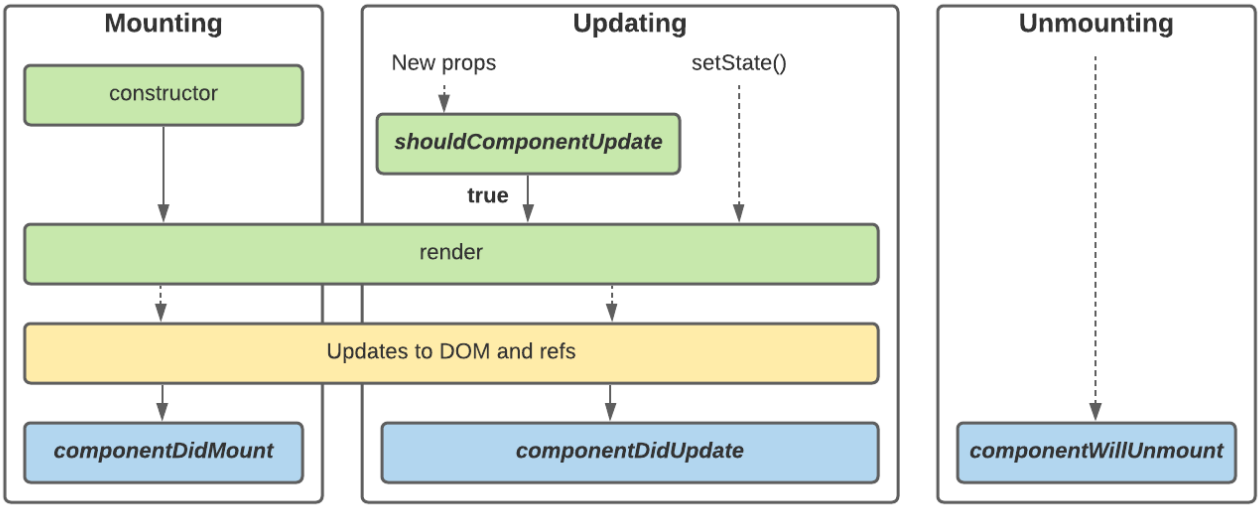
this.state = 250; //OR
this.state = [“asasasa”, “Sasasas”]; //OR
this.state = { name:”ASASAS” , AGE:30, AWARDS:[ ] } .

NOTE: It is a good practice to intialize state inside constructor.
We can capture some props and store it as a part of state.

Component LifeCycle
https://enlear.academy/a-deep-dive-into-react-lifecycle-methods-41e6acfd635a
constructor
componentWillMount
getDerivedStateFromProps
componentDidMount
render()

shouldComponentUpdate
getSnapshotBeforeUpdate
render()
componentDidUpdate

componentWillUnmount


Mounting
constructor
Updating
New props
setState()
shouldComponentUpdate
true
render
Updates to DOM and refs
componentDidMount
componentDidUpdate
Unmounting
componentWillUnmount

Event Handling

name in the Element should be camelCase

```
<p name="p1" id="p1" onMouseDown={this.handleSubmit} >Events Handling 1 in React</p>

The name of the function to invoke is in JS expression (that is in {} )
<p name="p1" id="p1" onMouseDown={this.handleSubmit} >Events Handling 1 in React</p>

The name of the function to invoke is in JS expression should be JUST THE NAME, that is without ().
CORRECT
<p name="p1" id="p1" onMouseDown={this.handleSubmit} >Events Handling 1 in React</p>
INCORRECT
<p name="p1" id="p1" onMouseDown={this.handleSubmit()} >Events Handling 1 in React</p>
```

If in class component, make sure to add this line in the constructor (that is bind the function/handler.) for every eventhandler we define. Like this.

```
this.handleSubmit = this.handleSubmit.bind(this);
```

React will always pass the event object automatically. We can use this event object to get information abbout the event.

```
handleSubmit(e) {
    e.preventDefault();
    alert('Type of event : ' + e.type);
    alert('Target of event : ' + e.target.id);
```

We can also pass extra information from our own side to the eventhandler. Like this.

```
<button type="submit" name="button2"
            onClick={(e) => this.handleEvent2(e, 'extra info')}>
            Submit Button 2 which passes extra params to the handler
</button>

handleEvent2(e, param2) {
    e.preventDefault();
    alert('Inside handleEvent2 Type of event : ' + e.type);
    alert('Inside handleEvent2 Target of event : ' + e.target.id);
    alert('Inside handleEvent2 Target of event : ' + param2);
  }
```

Form Handling

app\src\components\form-handling\ControlledComponentFormDemoWithMoreFormElements.js

NOTES: Have a state object that captures / sync with each of the form control.

```
this.state = {
```

```
      age: 0,
      qualification: "Msc",
      genderOptions: ["male", "female"],
      gender: "male",
      hobbiesOptions: ["Cooking", "Singing", "Cricket", "Tennis"],
      hobbies: [],
      errMsg: "errors will be shown here"
    }
```

Each of the form control should have an handleChange handler. Example:

```
Enter Qualification: <input type="text" name="qualification"
             value={this.state.qualification}
             onChange={this.handleChange}
          />
```

Write one single handleChange for all of the form controls.

html_css_js\Fetch-ajax-demo.html

```
////Fetch API for getting content
  sendAJaxRequest = () => {
    fetch("https://jsonplaceholder.typicode.com/users")
      .then(res => res.json())
      .then(res => logg.innerHTML = res[7].address.city);
  };

  //Fetch API for posting form content
  sendAJaxPostRequest = () => {
    fetch(
      "https://jsonplaceholder.typicode.com/posts",
      {
        method: "POST",
        body: JSON.stringify({
          userId: "1",
          title: document.forms["f1form"].n1.value,
          body: document.forms["f1form"].n2.value
        }),
        headers: {
          "Content-type": "application/json;chartset=UTF-8"
        }
      }
    )
      .then(res => res.json())
      .then(res => logg.innerHTML =
        `Response of posting. id=${res.id} title=${res.title}`);
```

```
};
```

Routing

<Route path=”/” element={<App1 />} >
<Route path=”/abc/p1” element={<Apple />} >
<Route path=”/profile” element={<UserProfile />} >
<Route path=”*” element={<NoSuchPage />} >

Path params

```
<Route path="/user/:city" element={<User />} />

Then the above route will match any urls like /user/1 user/2 …..user/asasa
```

Inside our User componet, we can get the value of the bcity path param like this:

```
const { name, city, stateOfCity } = useParams();
```

Search Params

If the user types something thing like /*user*/1?startdate=9&enddate=12

Then inside our component we can get the search paramters like this.

```
const [searchParams] = useSearchParams();
```

….

```
<br />Search params "season" has value: {searchParams.get("startdate")}
```

Link/Navlink

Is the anchor alternative in React.

Clicking on a Link will NOT REFRESH the page

Redux

provides us with a “store”, where application state be maintained in a centralised.

Store – is read-only
dispatch actions to the store to change the state stored in store.

Reducer functions in JS
state → newState

Reducer functions in Redux
(state , action)→ newState

where action is an object, which must an “type” attribute.

STEP 2: (Step 1 was to create reducer)
We create a store, by passing in the reducer(s)

```
var rootReducer = combineReducers(
  {
    reducer1: PlayerReducerUsingImmutableJSV2,
    reducer2: SecondReducer,
    reducer3: NewsReducer
  });

const store = createStore(  rootReducer )
```

STEP 3: Use the store in components.
STEP 3a: Create a container component that would connect the store to the presentation component (that display html content)
code mapStateToProps and mapDispatchToProps.
Connect them to the presentation component so that the presentation component gets the store data and the dispatch functions as props.

```
const mapStateToProps = (storeData) => {
  return{
    PlayersDataFromReducer1: storeData.reducer1,
    PlayersDataFromReducer2: storeData.reducer2
  }
}
```

```
const mapDispatchToProps = (dispatch) => {
  return{
    addDummyPlayerXYYZ: () =>  dispatch({ type: "ADD_ONE" }),
......
 }
}

export default connect(mapStateToProps,mapDispatchToProps)(ReduxDemoV2);
```

Full Stack Project (morning session+)

STEP-0: Backend ready (from your morning session)

STEP-1:
Create the Saga to watch action of type “ADD_NEW_PLAYER”
Saga should then call AddPlayer generator that will post the player information using the fetch post API to the back-end (on some URL say **/player/add**).

After Saga has posted the form content, it should put an action type “ADD_PLAYER_SUCCESS” so that reducer listening for this type, will update some state variable “add_player_success” to true and the UI can display the success message.

STEP-2: Code the reducer.

STEP-3: Add the reducer to the store.

STEP-4:
Create a new Component <AddNewPlayer> that has the form.
Connect it to the store, using mapStateToProps and mDToP.

SECOND PART:
USE CASE: In the same Add New Player, handle flow where a user with same name already exists in DB. In such case your backend should return a failure message. Based on this failure message, Saga will put a differement action type, which will then be acted upon by the reducer. Finally the UI component should display **NEW PLAYER ADD FAILED. Reason duplicate name.**

DAY 1	Back end ready	Git DB	DONE issue
DAY2	FE 1st use case		

	done
DAY 3	FE 2st use case
	done

Coding challenges 45 mins
Mock Interviews 1hr
Docker & Kubernetes concepts/command 30 mins

NodeJS/Express/Mongoose
GIT repo in GitHub

npm init -y
modify package.json express
npm install
Jest + Mocha + Chai
Postman

npx create-react-app
Header
Jest + Enzyme

App.js.
ComponentA -
Test reducers -

PROJECT Backend SERVER UNIT TESTS

npm install jest supertest

Add/ Modify following in package.json

"main": "app.js",

"type":"module",
"scripts": {
"test": "node --experimental-vm-modules node_modules/jest/bin/jest.js –coverage",
"start": "node app.js"

also: write your test cases in a folder src**__tests__**

Add this import if you get timeout issues
`import {jest} from '@jest/globals'`
Add this jest.setTimeout(15000); line inside describe but before test(….) line.

Describe ????

Refer Jest doc https://jestjs.io/docs/testing-frameworks
and from there https://alexanderpaterson.com/posts/how-to-start-unit-testing-your-express-apps

Test your UT like you do in postman:

```
describe('Testing Doctor Search', () => {
    jest.setTimeout(30000);

    test('neither zone nor volunteer valid', async () => {
        await request(app)
        .post('/doctors/search/name/Prashk')
        //.set('someparameter', 'somevalue')
        .send().expect(200)
    })
})
```

```
describe('Testing Doctor Add', () => {
    jest.setTimeout(30000);

    test('neither zone nor volunteer valid', async () => {
        await request(app)
        .post('/doctors/add')
        .set('name', 'somevalue')
        .set('age', 100)
         set('gender','somevalue')
        .send().expect(200)
    })
})
```

LOGGER

https://www.npmjs.com/package/simple-node-logger

```
npm install simple-node-logger --save

// Add this to your app.js and whereever you want logging.

const SimpleNodeLogger = require('simple-node-logger'),
        opts = {
                logFilePath:'mylogfile.log',
                timestampFormat:'YYYY-MM-DD HH:mm:ss.SSS'
        },
log = SimpleNodeLogger.createSimpleLogger( opts );
```

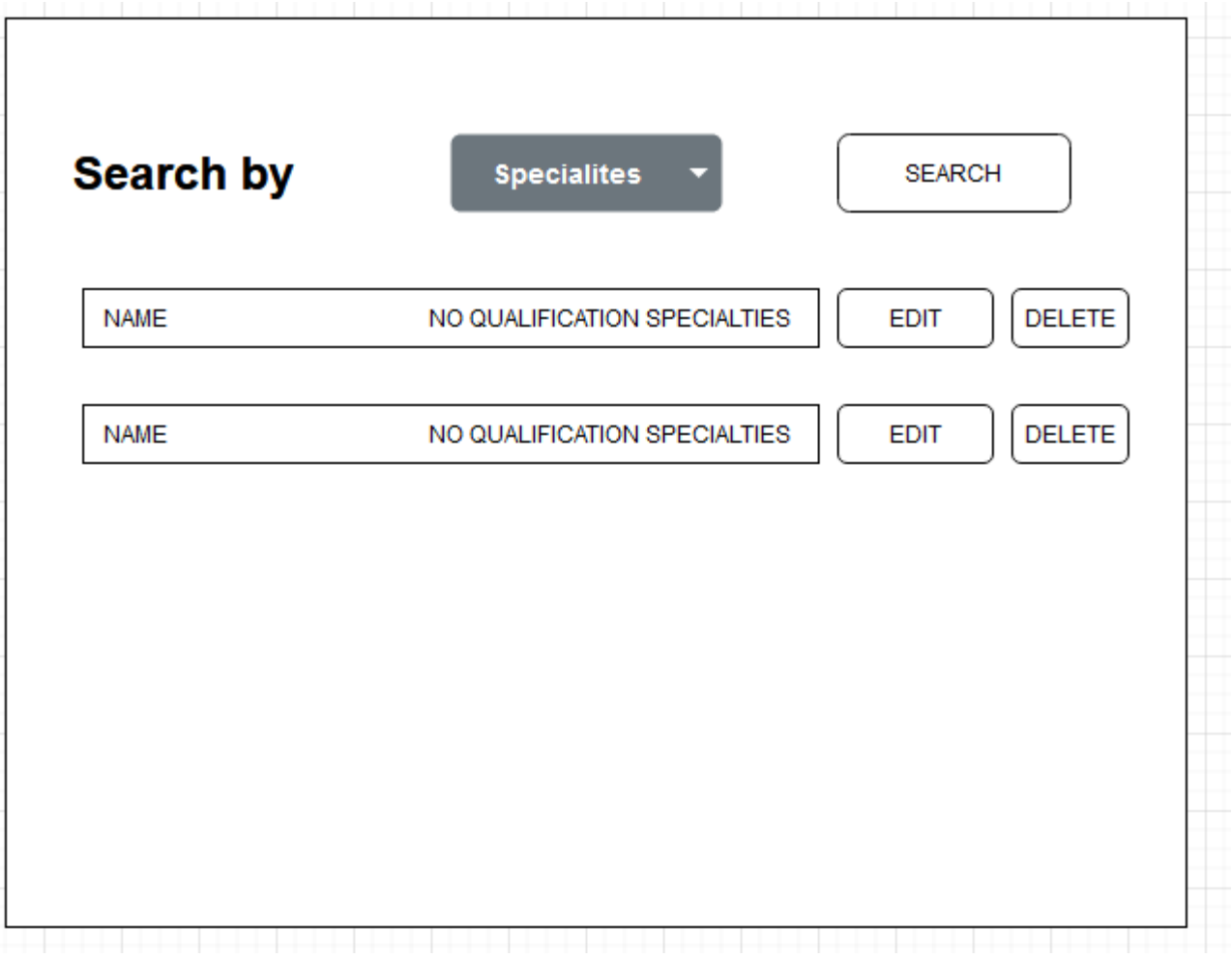
USAGE

```
log.info('subscription to ', channel, ' accepted at ', new Date().toJSON());
```

DAY-2 TASKS

TASK	COMPLETE BY	Duration	NOTES
Server Side UTs	11am	2hrs	+ comments if any
Ist User Story Search Mockups	11:30	30 mins	
Routes setup Header	12 noon	30 mins.	

Search Form	3 pm	90 mins	Form + state + component - 1hr Container - 30 mins
Two way of doing Redux-Saga or Direct Fetch using useEffect and Direct fetch for posting.	4Pm	1 hr	Saga - & Reducer
UTs	5pm	1hr	



,

<button onClick=”/appointments/delete/${appointments[0]._id}” > Delete </button>
Flow is almost similar to search user story.
.

TESTS

Setup and teardown

Like Mocha, Jest provides setup and teardown functionality for tests. Setup steps can be run before each or all tests using the **beforeEach()** and **beforeAll()** functions respectively. Similarly, teardown steps can be run after each or all tests with the **afterEach()** and **afterAll()** functions respectively.

The following pseudocode demonstrates how these functions can be used:

```
describe("test", () => {
  beforeAll(() => {
    // Runs once before all tests
  });
  beforeEach(() => {
    // Runs before each test
  });
  afterEach(() => {
    // Runs after each test
  });
  afterAll(() => {
    // Runs after all tests
  });
});
```

Mocking with Jest

Mocks enable you to test the interaction of your code or functions without having to execute the code. Mocks are often used in cases where your tests rely on third-party services or APIs, and you do not want to send real requests to these services when running your test suite. There are benefits to mocking, including faster execution of test suites and ensuring your tests are not going to be impacted by network conditions.

Jest provides mocking functionality out of the box. We can use a mock to verify that our function has been called with the correct parameters, without actually executing the function.

For example, we could change the test from the recipe to mock the **uppercase()** module with the following code:

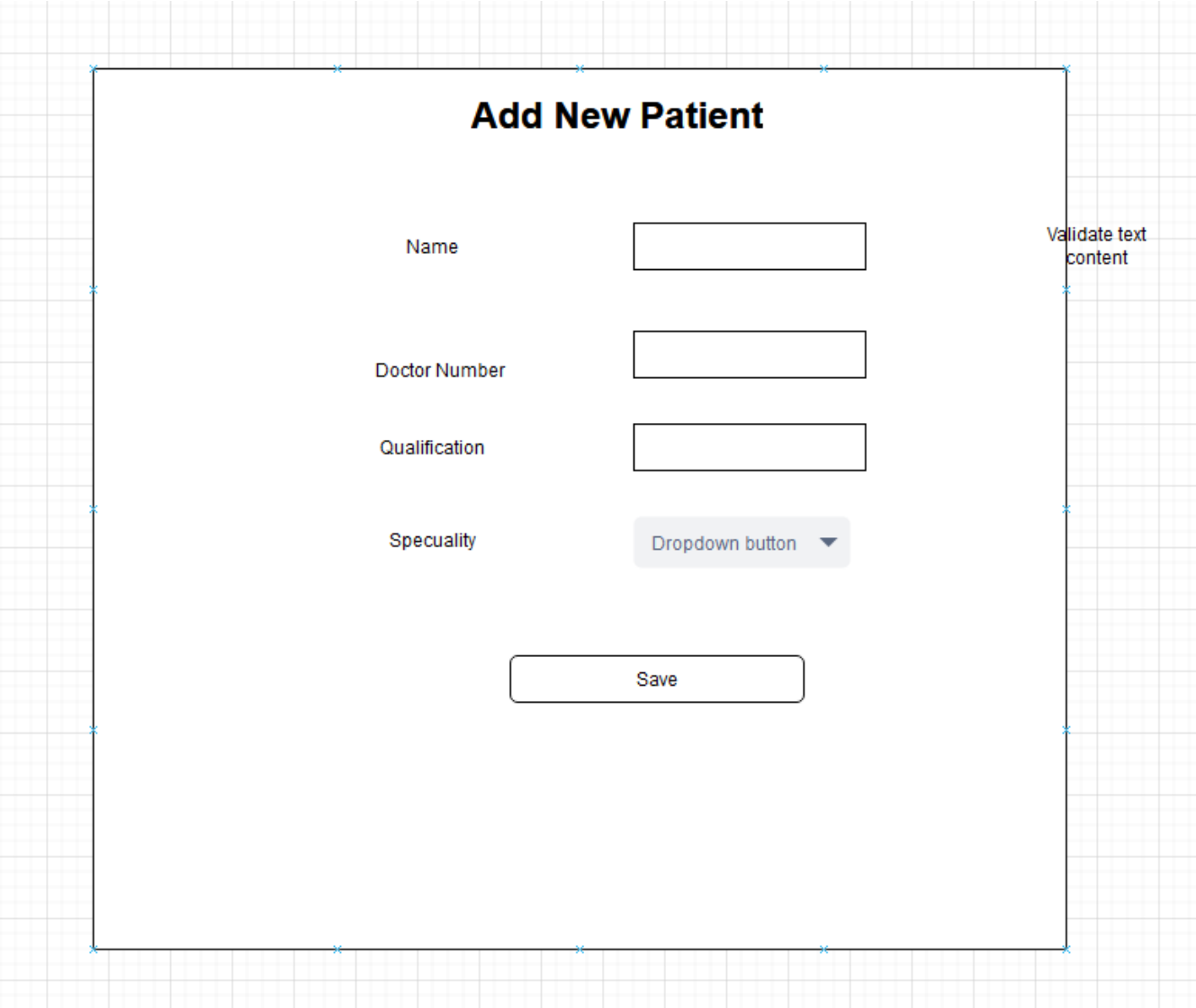
```
describe("uppercase", () => {
  test("uppercase hello returns HELLO", () => {
    uppercase = jest.fn(() => "HELLO");
    const result = uppercase("hello");
    expect(uppercase).toHaveBeenCalledWith("hello");
    expect(result).toBe("HELLO");
  });
});
```

jest.fn(() => "HELLO"); returns a new mock function. We assign this to a variable named **uppercase**. The parameter is a callback function that returns the string **"HELLO"** – this is to demonstrate how we can simulate a function's return value.

We use the **Expect** method **.toHaveBeenCalled()** to verify that our mock function got called with the correct parameter. If for some reason you cannot execute a function in your test suite, you can use mocks to validate that the function is being called with the correct parameters.

DAY-4 21-FEB-22 TASKS

TASK	COMPLETE BY	Duration	NOTES
Add Form UI	11am	2hrs	Presentation Component + UTs
Add Form UI	11:45	30 mins	Container Component +Add Route
Saga	12:30		
Reducer	1pm		
More Testing	3 pm		Functional + UTs

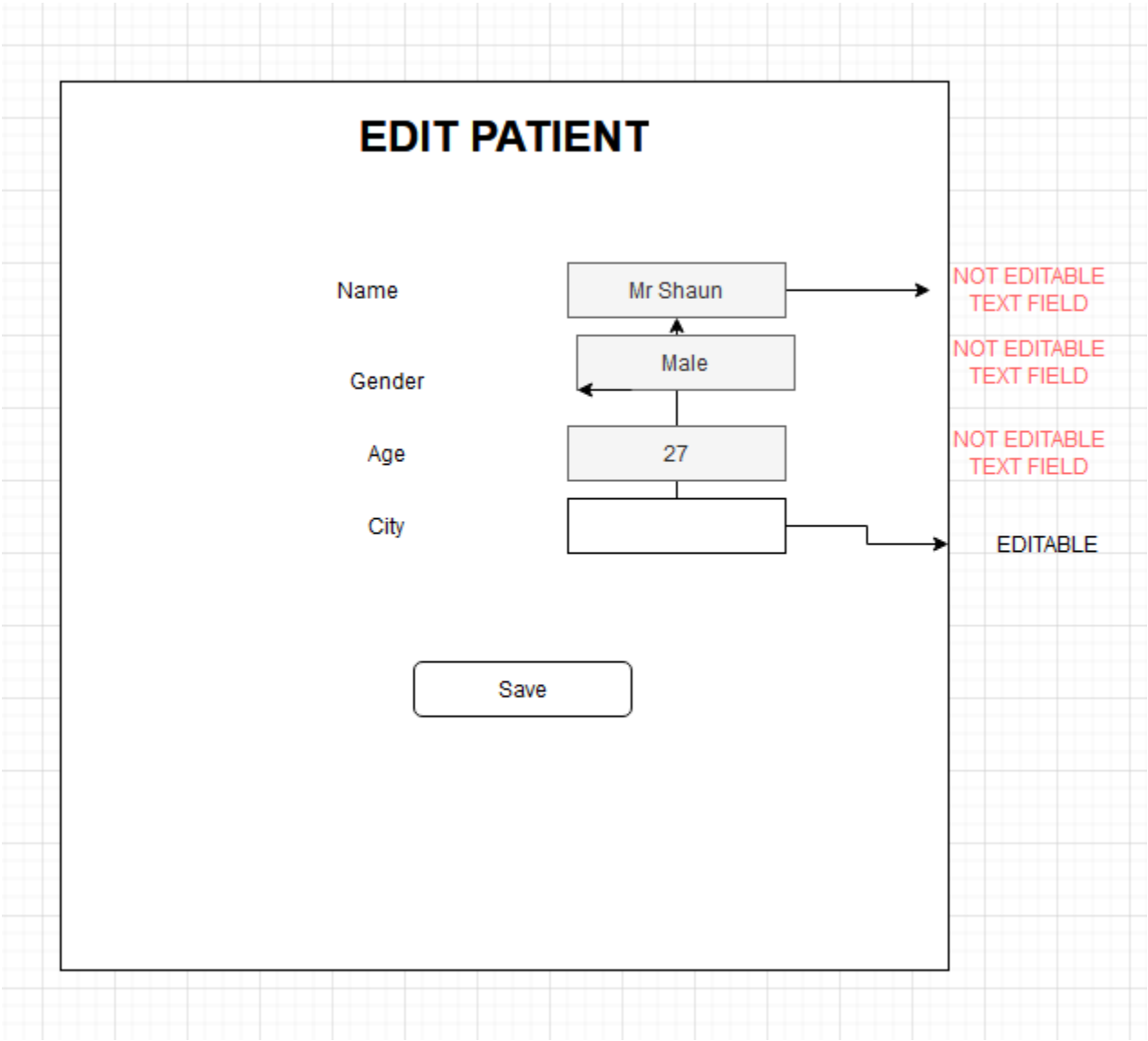


Psedo Code: Schematic

DAY-5 22-FEB-22 TASKS

TASK	COMPLETE BY	Duration	NOTES
Server Side Logs	10:30am	30 mins	
Edit Form UI			Presentation Component + Container Component +Add Route
Reducer	1pm		+UTs

More Testing	3 pm		Functional + UTs
Bootstrap Styling	4:30pm		



app.get("/patienthistories", (req, res) {
log.info ("Searching for patient histories based on " + req.params.fromdate);

Doctor.find

SUCCESS

log.info ("Found " + n + "results");

ERROR

```
        log.error (err );
        res.status(400).send("{msg: ……..}")

}
```

DAY-6 23-FEB-22 TASKS

TASK	COMPLETE BY	Duration	NOTES
Integration	11:30am	1 hr mins	
Bootstrap Styling	1pm		
More Testing			
Mock Interviews	3pm-6pm		
Flow diagram of your use case			

Integrate Backend first. Test it out using postman.
/doctors
/patients

React
Integrate component wise.
Integrate the reducers and sagas.
Store update.
Create appropriate routes in App.js
Top NavBar needs to be created

TIP:
Starter code for react bootstrap grid creation
https://codesandbox.io/s/reactbootstrapgrid-example-9m5p0?file=/src/index.js

Testing Components that use useLocation
https://testing-library.com/docs/example-react-router/

```
import {MemoryRouter} from 'react-router-dom'

test('full app rendering/navigating', () => {
```

```
  render(<App />, {wrapper: MemoryRouter})

  // verify page content for expected route
  expect(screen.getByText(/you are home/i)).toBeInTheDocument()
})
```

JS cheatsheet

```
<html>
<script>
  // If
  var x;
  if (x) { // here x is null so if condition will fail and it will NOT enter the bl
ock

  }

  // once it has a value, if will pass
  x = 10;// or x=true or x=new Car() or x={price:25}
  if (x) // will be taken as true. pass the condition


    // switch
    let x = "Wednesday";

  switch (x) {
    case "Monday": console.log("Haha first day of work week!"); break;
    case "Wednesday": console.log("Haha mid day of work week!"); break;
    case "Friday": console.log("Hurray!! last day of work week!"); break;
    case "Sunday": console.log("No work, All play!!!!"); break;
    default: console.log("Which day is this?? ");
  }

  // loop-1  using an counter
  let arr = [10, 20, 30, 40];

  for (let i = 0; i < 5; i++) {
    console.log(i);
  }

  /* loop-2 using forEach */
  arr.forEach(item => console.log(item));

  /* loop-3 using map */
  arr.map(item => console.log(item));
```

```
//OR if we also want the index of the item
arr.map((item, index) => console.log(`value at index ${index} is ${item}`));

//Array Filter function.
arr.filter(item => item <= 30); // ANSWER [10,20,30 ]

//Array includes function.
arr.includes(12);// false
arr.includes(20);// true

// Annonymous functions
// functions defined without a name
app.get("/doctors", (a, b) => { }) {
  // body of app.get
};

//Closures - are functions that are returned from another function

function parent(cash) {
  var twowheeler = 50000;

  return function child() {
    console.log(`I have ${cash}`);
  }
}


var child1 = parent(1000); // child1 is a function
var child2 = parent(2000); // child2 is a function

child1();// I have cash 1000;
child2();// I have cash 2000;

//Promises - are objects that do some work and may return the results in future.
// They are mostly used to do asynchronous work.
var myPromise = new Promise(function (resolve, error) { /* do some thing here */ })
;
//IF the job completyed successfully, promise will call the resolve function

//This is how we call the promise created above.
myPromise
  .then(result => console.log(result))
  .catch(error => console.log(error))

//Iterator and Generators
//Iterator - are functions that help us loop through array-like objects.
```

```
  var myRange = { from: 1, to: 10 };

  myRange[Symbol.iterator] = function () {

    return {
      next() {
        // apply some logic and return something if we want to, or say done=true;

      }
    }

  }

  //Example:
  myRange[Symbol.iterator] = function () {

    return {
      current: this.from,
      last: this.to,
      next() {
        if (this.current <= this.last) {
          return { done: false, value: this.current++ };
        } else {
          return { done: true };
        }
      }
    }

  }

  // Generators: are simplified iterators
  var myGenerator = function* (){
    yield 1;  // when somebody call this generator's next(), it will give this value
and PAUSE.
    yield 2;
  }

  var x = myGenerator().next().value ;// 1


  // Modules - are a way for authors to make some or all of the data and functions
  // defined in thier JS file(module) to become accessible in other JS/modules.
  // They use export to make them available in other JS.
  // Other JS/module use "import" to get access to the exported variable.

  //Modules help us in avoiding name collusions.
</script>
```

```
</html>
```