React JS

index.js

```
import React from 'react';
import ReactDOM from 'react-dom'

function Greeting() {
   return <h2> Hello </h1>
}

ReactDOM.render(<Greeting />, document.getElementById('root'))
```

↗ the component that you want to render

↳ the element where you want to render

JSX Rules

- return a single element (div / section / article / Fragment) <> </> or React.Fragment
- use camelCase for html attribute (like className, htmlFor, etc)
- write JS inside { }
- variables can be placed inside or outside functions

Props

```
const Book = (props) => {                    // or {title, author}
   return <div> {props.title} </div>
}                                  ↳ {title}

<Book title = 'xyz' />     // passing props.

{img, title, author: {name, age}}   // object destructuring

const BookDetails = {
    img: "   ",
    title: "   ",
    author: {
        name: "   ",
        age: "   "
    }
}
```

Props children

```
<Book img={BookDetails.img} title={BookDetails.title} >
    <p> Hello world </p>    // children
</Book>
```

```
const Book = ({img, title, children}) => {
  return (
    <div>{img}</div>
    <div>{title}</div>
    {children}        //accessing props children
  )
}
```

Notes:

- children → specific name
- children → (props)
- → children is not written here
- const {img, title} = props
- {props.children}

Storing objects in an array

```
const books = [
  {img: "    ",
   title: "    ",
   author: "    ".
  },
  {img: "    ",
   title: "    ",
   author: "    ",
  }
]
```

*

```
const BookList = () => {
  return <h1>{books}</h1>      //error
}
```

{books} → react cannot directly return objects.

```
const names = ['abc', 'def', 'ghi']
"   "   "
  return <h1>{names}</h1>      //abcdefghi
"
```

map(callback)

```
//accessing individual names using map()
const newNames = names.map((name) => {
    return <h1>{name}</h1>      // map must return something
})

{newNames}      // abc
                   de
                   ghi
```

```
*const BookList=()=>{
  return (
    <div>
      {books.map((book) => {
        const {img, title, author} = book
        return (
          <div>
            <h3>{title}</h3>
            <h6>{author}</h6>
          </div>
        )}
      )}
    </div>
)
}
```

```
<Book img={img}
      title={title}
      author={author}/>
```

OR

```
<Book book={book}/>
```

↳ passed as a prop

```
<Book book={book}/>
```

→ OR {book}

```
const Book = (props) => {
    const {img, title, author} = props.book
    :
}
```

* map should have a key

Previous example

```
{books.map((book) => {
    returns <Book key={book.id} book={book} />
})
}
```

Spread operator (...)

returns <Book {...book} />

↳ passes {img, title, author}

Google - Synthetic Events

Events

- Inline

onClick = {() => console.log(title)}

- Reference

onClick = {clickHandler}

const clickHandler = () => { alert('Hello world') }

- passing object

const clickHandler = (author) => { console.log(author) }

Eg: onClick = {clickHandler(author)}

↳ invoking the function

So, it is called authomatically without clicking

To avoid this, use arrow function:

onClick = {~~click~~() => clickHandler(author)}

* const clickHandler = (e) => {

    console.log(e)

    console.log(e.target) → the component that you're clicking

}

classmate
Date
Page

Imports and exports

- named export

```
export const books = [ ... ]
```

→ same name as the named export

```
import {books} from './books'
```

- export default

```
const Book = () => {
  return ...
}
export default Book
```

↳ same name as the function

* There can be only one export default but multiple named exports.

```
import Book from './Book'
```

↳ we can have any name and use it

useState (used when we want to re-render something)

Eg:

```
const ErrorExample = () => {
  let title = 'Random Title'
  const handleClick = () => {
    title = 'Hello People'      // on clicking the button, title value will
  }                             change but to return it, we have to
  return (                      re-render.
    <>
      <h1>{title}</h1>
      <button onClick={handleClick}>Change title</button>
    </>
  )
}
```

Hooks

- useState
- useEffect
- useRef
- useReducer
- useContext

```
import {useState} from 'react'

useState(a default value, function)   //syntax
                                  → returns an array [default-value, func]
```

↳ can be any value
- number
- string
- array
- object

```
const [text, setText] = useState('random title')   //array destructuring
                                                   destructuring

const handleClick = () => {
    setText('hello')   //this will re-render and change the title
}                      //but it will not revert back to 'random title'
                       on clicking the button again.
```

useEffect

- By default, useEffect runs after every re-render

* `useEffect(callback)` //syntax

- useEffect cannot be used inside conditional statements.

```
if (value > 0)
{
    useEffect(() => {
        console.log({value}, 'call useEffect')
    })
}
```

} Invalid

* `useEffect(callback function, dependency list)` //syntax

Eg:-
```
useEffect(() => {
    console.log('call useEffect')
```

```
if (value > 0)
    document.title = `New messages (${value})`.
}, [ ]);
```

↳ empty dependency list
(It will run only in the initial render)

```
useEffect (() => {
    ...
}, [value])
```

↳ Every time, the value changes, useEffect is called to re-render.

- There can be multiple useEffect.
- We cannot use async await inside useEffect as it returns a promise.

Fetch data

```
const [users, setUsers] = useState([])

const getUsers = async () => {
    const response = await fetch (~~de~~ url)
    const users = await response.json()
    setUsers (users)
}

useEffect (() => {
    getUsers()
}, [])
```

async-await function

```
useEffect (() => {
    fetch (url).
    .then ((resp) => resp.json())
    .then ((users) => {
```

- We can use fetch inside useEffect directly using .then .catch.

```
useEffect(() => {
  fetch(url)
  .then((res) => res.json())
  .catch((error) => console.log(error))
})
```
↳ network error
not 404 error

fetch does not catch 404 error. It catches only network error.

Short-circuit evaluations

text || 'abc' – If text is true, text will be displayed, otherwise 'abc'

text && 'abc' – If text is true, then 'abc' will be displayed, otherwise ~~text~~ nothing.

isError ? <h2> Error... </h2> : <h2> No Error... </h2>
↳ Ternary operator

Dynamic object keys

```
const name = e.target.name
const value = e.target.value
setPerson({...person, [name]: value})
```
↳ same as name: value
email: value
age: value
} instead of 3 separate inputs, use dynamic objects.

useRef

- preserves value
- does not trigger re-render
- targets DOM nodes / elements

```
const refContainer = useRef(null)    //syntax (similar to useState)
                       ↳ default value
ref = {refContainer}    //invoking useRef
↑ ref attribute
```

console.log(refContainer) → {current: null}
property ↳ our default value

<input type='text' ref={refContainer} />

After submitting, refContainer.current.value will have the value of input tag.

- The difference here is that we don't have to use value property and onChange everytime.

refContainer.current → points to the div element

useReducer

const [state, dispatch] = useReducer(reducer, defaultState) // syntax
state value — function — function (always added first) — default value for state (can be a variable or can be passed directly)

const reducer = (state, action) => { // reducer function
....
}
what action we want to perform on the state value

const defaultState = { // passed to the useReducer
people: [],
isModalOpen: false,
modalContent: '',
}

written instead of:
const [people, setPeople] = useState([])
const [isModalOpen, setIsModalOpen] = useState(false)
:

dispatch({type: 'TESTING'}) // using dispatch() to call reducer using action.type = 'TESTING'

- reducer function must always return state

```
const reducer = (state, action) => {
  if (action.type === 'TESTING') {
    return { ...state, people: data, isModalOpen: true,
                                     modalContent: 'item added'}
  }
  return state
}
```

```
dispatch ({type: 'ADD_ITEM', payload: newPerson})
```
↖ passing values through dispatch function

```
action.payload  // accessing payload
```

useContext (to avoid prop drilling)

- two components - Provider, Consumer

```
const PersonContext = React.createContext()   // syntax
```

- When we have a component tree that goes deeper, we can access functions ~~on~~ of the first component in the last component using useContext.

```
const contextAPI = () => {
  const removePerson = () => {
    ...
  }
  return (
    <PersonContext.Provider value={{removePerson}}>
      <h3> ... </h3>
      <List ... />
    </PersonContext.Provider>
  )
}
```

→ here, object; but can be any value

} wrap the component with PersonContext.Provider

```
const List = () => {
    return <SinglePerson />
}

const SinglePerson = () => {
    const { removePerson } = useContext(PersonContext)   // removePerson can be directly used without passing it down the tree.
    return (
        <button onClick={removePerson}> remove </button>
    )
}
```

Custom Hooks

Example : useFetch() → to fetch url.

- Custom hooks always return some value.
- They are used to make the code reusable.
- The returned values are destructured in the same way as for useState.

Naming convention - useCustom

Prop types

```
import PropTypes from 'prop-types'

const Product = ({image, name, price}) => {
    :
}

Product.propTypes = {          // same name as the component
    image: PropTypes.object.isRequired,    // types of the props; these props are required
    name: PropTypes.string.isRequired,
    price: PropTypes.number.isRequired,
}
```

```
Product.defaultProps = {
    name: 'default name',
    price: 3.99,
    image: defaultImage
}
```
→ import defaultImage from '../images/...'

this can be used

```
<p> ${price || 3.99}</p>
```

But if the property does not exist, we cannot use it.

```
image: {
    url: ...
}
```

```
{image.url}    // accessing url from image.
```

→ This will show error if image does not have url (as in the case of defaultProps)

```
const url = image && image.url
```
→ check if image is present
→ If image is present, check if url is present.

```
<img src={url || defaultImage} />
```
→ If image and image.url are present, then url will be used otherwise defaultImage.

React Router

```
npm install react-router-dom
```

```
import {BrowserRouter as Router, Route, Switch} from 'react-router-dom'
```
→ alias

```
return (
  <Router>
    <Route exact path = '/' >
      <Home />
    </Route>
    <Route path = '/about' >
      <About />
    </Route>
    <Route path = '/people' >
      <People />
    </Route>
  </Router>
)
```

this will show both the home page and about page.
To solve this, add exact

```
<Route path = '*' >
  <Error />
</Route>
```

matches every page

That means if we go to /hello which does not exist, error page will be shown.

The problem is if we go to /about or any other page, along with it, error page will be shown.

To avoid this, <Switch/> component is used. Only the first match is displayed.

```
<Router>
  <Switch>
    <Route exact path = '/' >
      <Home />
    </Route>
    <Route path = '/about' >
      <About />
    </Route>
    :
```

```
        <Route path='*'>
            <Error />
        </Route>
    </Switch>
</Router>
```

Link

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

<Link to='/about'>About</Link>
```

URL parameters

```
<Route path='/person/:id' children={<Person />}></Route>
```

:id → can be any name

/person/ → can be only /:id

:id → use the id to navigate.

In People Page,

```
<Link to={`/person/${person.id}`}>Learn More</Link>
```

useParams()

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

{id: "1"} → this will always be a string

useParam will have the id as object.

whatever is passed in the url patch path (here, id)

```
const {id} = useParams()
```

Node JS

- No DOM
- No window
- Server side apps

} Like in browser

Global variables

- __dirname - path to current directory
- __filename - file name
- require - function to use modules
- module - info about current module
- process - info about environment where the program is being executed.
- console.log()
- setInterval(), setTimeout, etc.

node app.js
node app

} to run a JS file

Common JS

- Every file is a module, by default.

~~app~~ names.js

```
const secret = 'SUPER SECRET'
const john = 'john'
const peter = 'peter'

module.exports = { john, peter }
```

whatever we want to export from names.js

func.js

```
const sayHi = (name) => {
    console.log(`Hello ${name}`)
}
module.exports = sayHi
```

~~app~~.js → {john: 'john', peter: 'peter'}

```
const names = require('./names~~.js~~')
```

names will be imported from names.js

```
const sayHi = require('./func~~.js~~')

sayHi('susan')
sayHi(names.john)
sayHi(names.peter)
```

OR

```
const {john, peter} = require('./names')
sayHi(john)
```

alternative.js

Alternative

```
module.exports.items = ['item1', 'item2']
```

// module.exports is an object
So, here, we set
exports = { items: [...] }

or

```
const person = {
    name: 'bob'
}
module.exports.singlePerson = person
```

```
const data = require('./alternative')
```

↳ { items: ['item1', 'item2'], singlePerson: { name: 'bob' } }

addition.js

```
const num1 = 10
const num2 = 15

function add() {
    console.log(`The sum is ${num1 + num2}`)
}

add()
```

app.js

```
require('./addition')
```

node app.js ⟶ The sum is 25

Built-in Modules

- OS
- path
- FS
- HTTP

OS

```
const os = require('os')
```
↳ using os, we can invoke any function in os module.

```
const user = os.userInfo()  // info about current user

console.log(`The system uptime is ${os.uptime()} seconds`)

const currentOS = {
    name: os.type(),
    release: os.release(),
    totalMem: os.totalmem(),
}
```

Path

```
const path = require('path')

path.sep  // returns the path separator (user/app)

const filePath =
path.join('/content', 'subfolder', 'test.txt') → /content/subfolder/test.txt
path.basename(filePath) → test.txt

const absolute = path.resolve(__dirname, 'content', subfolder', 'test.txt')
```
↳ gives the absolute path to test.txt.

FS (Sync)

```
const { readFileSync, writeFileSync } = require('fs')

const first = readFileSync('./content/first.txt', 'utf8')

writeFileSync('./content/result.txt', `Hello world`, {flag: 'a'})
```

'utf8' → appends to the end if the file exists.

writeFileSync → creates a new file if not present, otherwise overwrites the existing file.

FS (Async) → readFile, writeFile

```
readFile('./content/first.txt', 'utf8', (err, result) => {
    if (err) {
        console.log(err)
        return
    }
    console.log(result)
})
```

callback function is required in async FS module.

readFile(path, encoding, callback)

To use result in another readFile

```
console.log('start')
readFile('./content/first.txt', 'utf8', (err, result) => {
    if (err) {
        ...
    }
    const first = result
    readFile('./content/second.txt', 'utf8', (err, result) => {
        ...
        ...
        ...
        const second = result
        writeFile('./content/result-async.txt', `The result is ${first}, ${second}`, (err, result) => {
            if (err) {
                ...
            }
```

```
    console.log('done with the task')
  })
})
console.log('starting next task')
```

In sync fs, all the tasks are done sequentially.
In async fs, the control can go do other tasks while reading files which takes time.

In case of sync,

Output

start
done with the task
starting new task

In case of async,

Output

start
starting next task
done with the task

Instead of waiting for one task to be completed, it does other tasks at the same time.

HTTP

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

const server = http.createServer((req, res) => {
  res.write('Welcome')
  res.end()
})

server.listen(5000)
```

req: incoming request (that the client sends)
res: the response (that the server sends)
res.end() → ends the response after it is sent
server.listen(5000) → the server opens up at port 5000
5000: port number

localhost:5000 → Welcome

shorthand

```
res.end('Welcome')
```

```
const server = http.createServer((req, res) => {
    if (req.url === '/') {
        res.end('Welcome to home page')
    }
    else if (req.url === '/about') {
        res.end('About page')
    }
    res.end(`
    <h1>Oops!</h1>
    <a href='/'>Back home</a>
    `)
})
```

we can have html

NPM

npm - global command (installed with node)

npm i <packageName> // local dependency - we can use it only in this particular project

npm install -g <packageName> // global dependency - we can use it in any project

npm init - to create package.json, step-by-step ('enter' to skip)
npm init -y - to create package.json with default values.

Using dependencies

~~cmd~~ npm i lodash

```
const _ = require('lodash')
```

.gitignore

/node_modules

Dev dependencies

npm i nodemon -D
or
--save-dev

scripts in package.json

```
"scripts": {
    "start": "node app.js"      // npm start will run node app.js.
    "dev": "nodemon app.js"     // npm run dev
}
```

→ nodemon will watch our app.js and restart whenever we save our changes automatically

FS (Async) - Setup Promises

~~const fs = require('fs')~~
const {readFile} = require('fs')
const getText = (path) => {
    readFile(path, 'utf8', (

FS (Async) - Using Promises

```
const {readFile} = require('fs')

const getText = (path) => {
    return new Promise((resolve, reject) => {
        readFile(path, 'utf8', (err, data) => {
            if (err) {
                reject(err)
            }
            else {
                resolve(data)
            }
        })
    })
}
```

```
getText('./content/first.txt')
  .then((result) => console.log(result))
  .catch((err) => console.log(err))
```

This will still be completed if we had to use the result for another work. We will have to nest the tasks.

To avoid this, use async-await.

```
const start = async () => {
    try {
        const first = await getText('./content/first.txt')   // wait for the promise to get resolved.
        console.log(first)
    }catch(error){
        console.log(error)
    }
}
```

getText() will only do the task of readFile.
We have to write another function for writeFile.

Instead of that, we can use promisify().

```
const { readFile, writeFile } = require('fs')
const util = require('util')
const readFilePromise = util.promisify(readFile)
const writeFilePromise = util.promisify(writeFile)

const start = async () => {
    try {
        const first = await readFilePromise('./content/first.txt', 'utf8')   // we can use readFile as it also returns a promise
        const second = ''       ''
```

writeFile

```
await writeFilePromise('./content/result.txt', `The result is: ${first}, ${second}`)
} catch(error) {
  console.log(error)
}
}
```

Event Emitters

```
const EventEmitter = require('events')
const customEmitter = new EventEmitter()
```

any name

a string (name of the event)

example

```
customEmitter.on('response', () => {
  console.log('data received')
})

customEmitter.emit('response')
```

same name

- on - listen for an event
- emit - emit an event.

We can have any number of same ~~#~~ events with different callbacks.

```
customEmitter.on('response', (name, id) => {
  console.log(`data received user - ${name} with id: ${id}`)
})

customEmitter.emit('response', 'john', 34)
```

other parameters can be passed.

Another way to interact with server

```
const http = require('http')
const server = http.createServer()

server.on('request', (req, res) => {
    res.end('Welcome')
})

server.listen(5000)
```

Streams - reads data in small chunks.

```
const {~~read~~ createReadStream} = require('fs')

const stream = createReadStream('./content/big.txt', 'utf8')  ~~{highWaterMark}~~

stream.on('data', (result) => {
    console.log(result)
})
```

Total - 169 KB
returns 64KB buffer, 64KB buffer, and then remainder (last buffer)

- returns 64K bytes of data, by default.
- control size - highWaterMark
  → `const stream = createReadStream('./content/big.txt', {highWaterMark: 90000})`
- {encoding: 'utf8'}

```
stream.on('error', (error) => {      //error
    console.log(error)
})
```

Reading ~~file~~ text files using ~~fs~~ createReadStream is more useful than using readFileSync.
→ this reads the whole file at a time.

Status Code

200 - the request was successful
404 - error (not found)
100 - request error
401 - unauthorised request

Example - writing data / sending response in chunks.

```
var http = require('http')
var fs = require('fs')

http.createServer((req, res) => {
    const fileStream = fs.createReadStream('./content/big.txt', 'utf8')
    fileStream.on('open', () => {
        fileStream.pipe(res)    // pipe(res) pushes the readStream into writeStream. So, data is read in chunks and also written or sent in res in chunks.
    })
    fileStream.on('error', (err) => {
        res.end(err)
    })
}).listen(5000)
```

Headers

Content-type → text/html // returns html in the response
→ application/json // returns application js