||Hare Krishna||

Miracle
Pg.No :
Date: / /20

# React

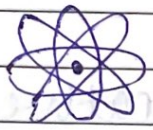
* Create a HTML page (index.js)

```
import React from "react"
import ReactDOM from "react-dom"

const page = (                    → JSX should always be nested under a single parent element
    <div>                                          style
        <img src="./__.png" width="40px" />        add closing tag in a
        <h1> Fun facts ... </h1>
        <ul>                                       children elements
            <li> ___ </li>
            <li> ___ </li>
            .
            .
            .
        </ul>                     → Same as HTML
    </div>
)

ReactDOM.render(page, document.getElementById("root"))
   su render karwa de        kya aagad render karwa de
```

→ index.html

```
<html>
  .
  .
  .
  <body>
      <div id="root"></div>
      <script src="index.pack.js"></script>
  </body>
</html>
```

→ Output : (index.html)

 → image

Fun Facts..... → h1

- :
- :
- :
- :

will

✱ React is composable and declarative.

✱ Quiz :

1) Why do we need to 'import React from "react"' in our files?

→ React is what defines JSX.

2) If I were to console.log(page) in index.js, what would show up?

→ A javascript object. React elements that describe what React should eventually add to the real DOM for us.

3) What's wrong with this code:

```
const page = (
    <h1> ___ </h1>
    <p> ___ </p>
)
```

→ We need our JSX to be nested under a single parent element.

4) What does declarative and imperative means?

→ Declarative : I can tell the computer WHAT to do and expect it to handle the details.

Imperative : I need to tell it HOW to do each step. (Like in JS)

5) What does 'composable' means?

→ Small pieces we put together to make something larger / greater than the individual pices.

✱ Custom components

→ PascalCase : First letter of each word is capital.

→ camelCase : First letter small, else capital.

→ Components has PascalCase convention.

→ Custom components : Independent pieces of functionality that you can reuse in your application, and are the building blocks of all React applications.

→ They are simple JavaScript functions and classes, but you can use them as if they were customized HTML elements.

→ Function component :

e.g.
```
function Page() {
  return(
    <>
      html code
    </>
  )}
```

single parent element ←

This is called a component

JSX

```
ReactDOM.render(<Page />, doc.getElementId(" "))
```

★ Quiz

1) What is a react component ?

→ A function that returns React elements (UI)

2) What's wrong in the code ?

```
function myComponent(){
  return(
    <small>___ </small>
```

```
    )
}
```

→ Pascal Case in function name (component name) i.e. MyComponent()

3) What's wrong with the code?

```
function Headr() {
    return (
        <header>
        :
        </header>
    )
}

ReactDOM.render(Headr(), doc.get... (""))
```

→ Should be like HTML tag in components i.e. < Headr />

✱ Parent / Child Components

e.g

```
function Headr() {
    return (
    )
}

function Footr() {
    return (
```

```
    )
}
```

→ Parent component

```
function Page() {
    return(
        <div>
            <Header />  } Child components
            <Footer />  }
        </div>
    )
}
```

ReactDOM.render(<Page />, doc...)

✱ Styling with classes

e.g. <ul className="nav-items">
:
</ul>

→ in style.css file
.nav-item {
}

Instead of class, we use className in react.

✱ Organize components

→ import and export.

→ Create new files for each component.

e.g. Header.js → import React from "react" in all files where we use JSX

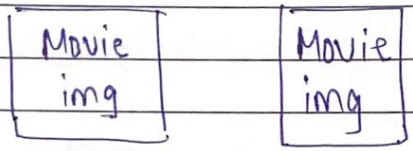
export default function Header() { ... }

→ index.js

import Header from './Header'

✱ React is data-driven

e.g. Netflix page




Mostly in all websites like Amazon, Netflix this type of component keeps getting repeated. And the data also is not hard-coded.

The data is stored in database and we just have to add img, name etc to it. So, if we do it statically we have to keep creating components again and again.

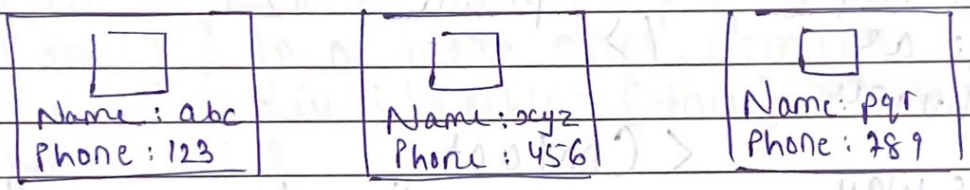
So, it can be said that it is not reusable. Here, comes the PROPS.

So, we can create a component and pass some parameter. In this way it can be reusable.

## Props:

Props are arguments passed into React components.

e.g. Output:




Code (using props)

→ contact.js

argument

```
export default function Contact(props) {

  return (
    <div>
      <img src={props.img} />
      <h3>{props.name}</h3>
      <p>{props.phone}</p>
    </div>
  )
}
```

→ App.js

```
import Contact from "./Contact"
```

⋮

```
export default function App() {
    return (
        <div>
            <Contact
                img = "./images/abc.png"
                name = "Abc"
                phone = "123"
            />
            <Contact
                img = ".../xyz.png"
                name = "xyz"
                phone = "456"
            />
            <Contact
                img = "   "
                name = "pqr"
                phone = "789"
            />
        </div>
    )
}
```

props. — je name aapiyu hoi ae lakhvanu.

In this way, we can use the same 1 function for 3 different arguments. Same is done in Netflix, Amazon, Youtube for reusable components

## ✱ Prop quiz

1) What do props help us accomplish?

→ Props passed into a ~~function~~ component helps us make a component more usable

2) How do you pass a prop into a component?

→ <MyHeader title="Khushi" />
Component prop

3) Can I pass a custom prop (e.g. 'blahbla={true}') to a native DOM element (e.g. ) Why or why not?

→ No.
Because the JSX we use to discribe the native DOM elements will be turned into Real DOM elements by React. And real DOM elements only have the property / attributes specified in the HTML specification (which doesn't include properties like 'blahbla')

4) How do I receive props in a component?

→ function Navbar (props) {
return (
:
)}

5) What datatype is 'props' when the component receives it?

→ An object!

→ Another way of writing props.

```
function Contact({img, name, phone}){
    return(
        <div>
            <img src={img}>
            <h1>{name}</h1>
            <p>{phone}</p>
        </div>
```

directly into func. but name should be same as defined whil passing prop

```
<Contact
    img=
    name=
    phone=
/>
```

✳ Array.map()

map() method creates a new array populated with the results of calling a provided function on every element in the calling array.

e.g. 1) const nums = [1, 2, 3, 4]
Output : [1, 4, 9, 16]

↳ const squares = nums.map(x => x * x)
console.log(squares)

2) const names = ["khushi", "patel"]
Output : ["Khushi", "Patel"]

↳ const capital = names.map((name) => {
    return name[0].toUpperCase() + name.sli
})

function () => {

Miracle
Pg.No :
Date: / /20

3) const pokemon = ["khushi", "Patel"]
Output : [" khushi ", " Patel "]
↳ const para = pokemon.map(mon => `<p>${mon}</p>`)

If one single line in return, we can also write this way

✱ Event Listeners (Handling Events)

→ camelCase

→ With JSX, we pass a function as the event handler, rather than a string

- e.g. In HTML,
  no camelCase
  <button onclick="activate()">
  Activate
  </button>
  a string
- In React
  camelCase
  <button onClick={activate}>
  Activate
  </button>
  function

→ There are many more event listeners in React. See documentation.

Miracle
Pg.No :
Date: / /20

## State

⟹ "PROPS" refers to the properties being passed into a component in order to work it correctly, similar to how a function receives parameters (from above)

A component receiving props is not allowed to modify those props i.e. they are immutable.

e.g. function A(a, b) {
return a+b
}
A(1,2)
= 3

function Add(a, b) {
a = 42
return a+b
}
console.log(Add(1,2))
Output : 44

Even if we added a=1, in the func. it is set to 42. that is wrong. We are not allowed to modify it (same in props)

⇒ "STATE" refers to the values that are managed by the component, similar to variables declared inside a function.

Anytime you have changing values that should be saved / displayed, you'll likely be using state.

⇒ Prop vs. State Quiz:

1) How would you describe the concept of "state"?

→ A way for React to remember saved values from within a component. This is similar to declaring variables from within a component, with few added bonuses.

2) When would you want to use props instead of state?

→ Anytime you want to pass data into a component so that component can determine what will get displayed on the screen.

3) When would you want to use state instead of props?

→ Anytime you want a component to maintain some values from within a

component. (And "remember" these values even when React re-renders the component)

4) What does "immutable" mean? Are props immutable? Is state immutable?

→ Unchanging
Props are immutable (like they are mutable if we try to set it to somethin else, but you should not change props)

State is not immutable. (i.e. its changing)

⟹ useState → is a ~~hook~~

import React, {useState} from "react"
→ Agar aai declare Kariye to niche React lakhvani nai jarur.

or React.useState()

↳ e.g

```
const result = React.useState()
consol.log(result)
```
↳ [undefined, f()] → We get an array one undefined and other f() is func

```
const result = React.useState("Hello")
console.log(result)
```
↳ ["Hello", f()]

→ Destructuring ~~useArray~~ useState

```
const [result, func] = React.useState("Hi")
```

result ↑ value of state (undefined)
func ↑ function (f())

```
console.log(result)
```
=> Hi

→ Changing state

The function allows us to make changes so that React can handle the changes accordingly.

→ Example

Create a counter:

```
const [count, setCount] = React.useState(0)

function Increment() {
    setCount((prevCount) => {
        return prevCount + 1
    })
}

function Decrement() {
    setCount(prevCount => prevCount - 1)
}
```

Callback func.

One line function return value hai toh aaru lakhai direction without { } etc.

html:

<button onClick={Increment}> + </button>

<h1> {count} </h1>

<button onClick={Decrement}> - </button>

*/ Note :

Whenever you need the old value of state to determine the new value of state, you should pass a callback function to your state setter function instead of using state directly.

This callback function will receive the old value of state as its parameter, which you can then use to determine your new value of state.

⟹ Changing state quiz :

1) What are the two ways to pass in a state setter func. (e.g. setCount)

→ (1) New value of state ( setCount(42) )

(2) Callback func.
Whatever callback func. returns ==> new value of state.

2) When to use option (1)?

→ Whenever you don't need the previous value of state to determine what the new value of state should be.

3) When to use option (2)?

→ Whenever you DO need the previous value to determine the new value.

✱ About hooks (Web Dev Simplified)

→ useState is a hook.

→ We cannot use hooks inside a class component. Only inside functions we can use.

→ We cannot use it inside if conditions, for loops (inside any nested condition)

→ Call it in order.

e.g. function App() {
useState() → 1st aa call thase
useState() → 2nd aa (1st pachi j)
}

Miracle
Pg.No :
Date: / /20

# Coursera - Advanced React

* Transforming lists in Javascript

→ By .map() method.

e.g. I have data.js file that has an array of objects of a dissert menu.

```
const data = [
    {
      id: "1"
      title: "Ice Cream",
      discription: " ",
      price: " ",
      image: " "
    }
    {
      .
      .
      .
]
```

Now, I only want the title, discription and price to show on my page. Do it using .map() method.

```
const desserts = data.map(dissert => {
    return {
      content: `${dissert.title} - ${dessert.
                                    discription}
      price: {dissert.price}
    }
})
```

→ The map() method returns a new array.

→ The map() method is useful for handling third party data.

→ The map() method is a transformation operation.

✡ Render a simple list component

```
const list = data.map (dessert => {
    return <li> {dessert.title} </li>
})
```

In code:
```
<ul>
    {list}
</ul>
```

✡ Keys

→ What is the need of keys?

e.g Add element to list

```
<ul>
    <li> Beer </li>
    <li> Wine </li>
</ul>
```

If we add a new item at the end of the list, React will work well. It will

match the two Beer trees, two Wine trees and add the third new Cider.

But what if we want to add a new item at the beginning?

React will show worst performance because React will mutate every child instead of realizing it can keep the beer and wine intact. This inefficiency can be a problem.

To solve this issue, react supports key attribute.

→ What are keys?

Keys are identifiers that help React to determine which items have changed or added or are removed.

Also instruct treatment of elements when updates happen.

Use stable identifiers that is always unique among its siblings.

→ Prevents key collisions

① Math.random() : Does not preserve internal state of list items

Why?

Miracle
Pg.No :
Date: / /20

Because when re-rendering occurs, those keys will be different, resulting in React having to recreate your list from scratch.

(2) Item Index : Indexes are not recommended for keys if the order of items may change, for eg. in cases where your list has sorting capabilities or users can either add or remove items.

Why?
Negatively affect performance
User interface glitches

(3) Unique ID : e.g. UUID()

Miracle
Pg.No :
Date: / /20

Because when re-rendering occurs, those keys will be different, resulting in React having to recreate your list from scratch.

(2) Item Index : Indexes are not recommended for keys if the order of items may change, for eg. in cases where your list has sorting capabilities or users can either add or remove items.

Why?
Negatively affect performance
User interface glitches

(3) Unique ID : e.g. UUID()

Day-2

* Forms in React

* What are controlled components?

→ Controlled components are a set of components that offer a declarative API (Application Programming Interface) to enable full control of the state of form elements at any point in time using React state.

→ How?

Using value prop → state delegation is performed via value prop

Value → A special property to determine input content at any time during under life cycle.

→ How to create a controlled component?

1. Local State (useState)
2. Value prop
3. onChange callback

onChange callback → Receives an event parameter, which is an event object representing the action that just took place similar to events on DOM elements.

To get the new value on every keystroke, we need to access the target property from the event and grab the value from that object, which is a string.

```
handleChange (event) {
    setValue(event.target.value)
}
```

→ When form is submitted ? (What to do)
How

```
<form onSubmit={handleSubmit}>
    .
    .
    .
</form>

handleSubmit(event) {
    validate(value) ——→ validating input values
    event.preventDefault()
}
```

To have control over the form values when form is submitted, use onSubmit prop in <form> tag.

onSubmit callback receives event parameter

```
<form onSubmit={handleSubmit}>
    :
</form>

handleSubmit(event) {
    validate(value)  → validating input values
    event.preventDefault()
}
```

To have control over the form values when form is submitted, use onSubmit prop in <form> tag.

onSubmit callback receives event parameter

Day-3

* Controlled vs. Uncontrolled Components

→ Uncontrolled Components

↳ Inputs are like standard HTML input.

↳ How to get value? (of the input typed)

↓

Using a React ref. → pull the value from the field when needed.

e.g.

```
const Form = () => {
    const inputRef = useRef(null)
```

```
const handleSubmit = () => {

    const inputValue = inputRef.current.value
    // do something with the value
}

return (
    <form onSubmit={handleSubmit}>
        <input ref={inputRef}
               type="text"
        />
    </form>
)
}
```

→ Mostly, try to use Controlled Components.

→ `<input type="file" />`

↓

always an uncontrolled component. because its value is read-only and can't be set programmatically.

e.g.
```
<form onSu={hanSub}>
    <input type="file" ref={fileInput} />
</form>

const fileInput = useRef(null)

const hanSub = (e) => {
    e.preventDefault()
    const files = fileInput.current.files
}
```

Miracle
Pg.No :
Date: / /20

| Controlled C. | Uncontrolled C. |
| --- | --- |
| ↓ | ↓ |
| Component's state handles form data | DOM handles form data |

→ Events:

onBlur(): Triggered when the user moves the cursor away from the element.

onFocus(): Triggered when element receives focus (cursor on the element) Opposite of onBlur event.

✡ React Context

| Props | State |
| --- | --- |
| ↓ | ↓ |
| Passed to the component | Managed within the component. |

→ Similarities bet^n props and state:

1) Plain JS objects
2) Deterministic (this means the component always generates the same output for the same combination of props and state)
3) Trigger render updates (both changes on trigger or render)

→ Props:

- Properties (full form)
- Component's configuration
- They are received from parents in the tree
- Immutable
- A component cannot change its props but it is responsible for putting together the props of its child components.

→ State:

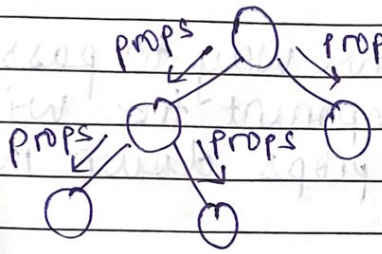
- This object is a way to allow react to determine when it should re-render.
- Has a default value
- Private

Components

| Stateless | Stateful |
|---|---|
| • Props only, no state | • Both props and state |
| • Easy to follow and test | • Charge of client-server communication, data processing and responding to user events |

## ✡ What is Context, and why is it used?

→ In a typical React app, data is passed from parents to children via props in a top-down fashion.




→ However, there are certain type of data that is needed by many components within an app.

→ In this case, using props is not always effective.

→ So, the alternative way of passing data is called Context

e.g. Theme switcher (dark-light mode)

→ Props drilling problem

Passing data through all component tree levels (even if components do not need it) i.e. passing through props.

Miracle
Pg.No :
Date: / /20

Parent component has to drill down props all the way to the children that need to consume them.

→ How to solve this problem?

Context API

→ Provides alternative way to pass data through the component tree without having to pass props down manually at every level.

→ Useful for global state

e.g. Website:

Blog App

Hello _____

What is Computer?

⋮ } Blog

Written by _____.

Here, both the blanks should have the name of the current logged in user.

Miracle
Pg.No :
Date: / /20

→ Code :

Components : <Header /> → title & hello
<LoggedInUser /> → user authentication
<Page />
↳ Blog name title
Blog
Written by ____

Hello ____ → <LoggedInUser/> ma lakhelu che.
i.e.

```
function LoggedInUser() {
    <h3> Hello .... </h3>
}
```

So, basically Header func. ma Blog App title and <LoggedInUser /> che.

→ Now, 2 components need to know about authenticated user name.
① LoggedInUser
② Page

→ Because an authenticated user falls into the nature of global data, shared across several components, that needs to be this is a clear example where context is perfect tool for the job.

① Create new file "UserContext.js"

```
import { createContext } from "react"
```

function that gives you a new context object back.

```
const UserContext = createContext(undefined)
```

func. value is a default argument here we set it to undefined

Provider comp.

```
export const UserProvider = ({children}) => {
    return <UserProvider>
    return <UserContext.Provider value={{user}}></UserContext.Provider>
```

has a value prop.

```
    const [user] = useState({
        name: "John",
        email: "john@gmail.com",
        dob: "01/01/2002",
    })
}
```

just for eg. we have added hard coded data to understand.

```
export const useUser = () => useContext(UserContext)
```

To provide a way for components to subscribe to the context,

Create a custom hook that wraps the useContext() hook

↓

A way to consume a context value.

→ Now, wrap <App/> comp. in App.js with <UserProvider />

```
<UserProvider>
    <App />
</UserProvider>
```

→ Now, to use the user details in <LoggedInUser> and <Page> comp,

```
func. Logg... () {
    const {user} = useUser()
    return (
        <p> Hello {user.name} </p>
```

Same for <Page /> component.

Miracle
Pg.No :
Date: / /20

Day - 4

# How re-rendering works with Context

→ When a component consumes some context value and the value of this context changes, that component re-renders.

→ When it comes to the default behavior of React rendering, if a component renders, React will recursively re-render all its children regardless of props or context.

e.g. Top level component injects a context provider at the top.

⇒
```
function App() {
return (
    <AppContext.Provider>
        <CompA />
    </AppC.Pro.>
)
}

const CompA = () => <CompB />
const CompB = () => <CompC />
const CompC = () => null
```
} functions. same as `func CompA(){...}`

Now, if App re-renders, A, B, C all three will re-render as well.

App(Context Provider) → A → B → C

Miracle
Pg.No :
Date: / /20

If the components are complex in nature, this could result in performance hit.

→ How to solve this issue?

⇓

React.memo() & useMemo hook

✱ React.memo()

→ If your component renders the same result given the same props, you can wrap it in a call to React.memo() for a performance boost by memoizing the result.

→ Memoization is a programming technique that accelerates performance by caching the return values of expensive func. calls.

→ This means that React will skip rendering the component, and reuse the last rendered result.

✱ useMemo()

→ Used for caching a specific calculation or value

accepts 2 arguments

→ const memoized = useMemo(compute, dependencies)

(More about hooks in next module)

Miracle
Pg.No :
Date: / /20

Day-5

☆ Working with React hooks

→ Hooks are Javascript functions that manage the state behavior and side effects by isolating them from a component.

→ We can isolate all the stateful logic in hooks and use them into the components.

☆ useState hook

(From Scrimba notes)

☆ useEffect hook

⟹ What are side effects?

→ A side effect is something that makes a function impure.

| → Pure function | Impure function |
| --- | --- |
| • No side effects | • Has side effects |
| • They receive an input and produce a predictable output of JSX. | • Not predictable because they are actions which are performed with the "outside world." |

- Always return the same output (given the same input)
- Predictable, reliable and easy to test.

- Side effects include:
  1) invoke console.log
  2) invoke fetch
  3) invoke geolocation,
  4) interact with browsers API
  5) Request to an API

etc.

→ In simple words,

Side effect is something external or outside of a function.

→ Why useEffect() ?

It provides a way to handle side effects.

useEffect is a tool that lets us interact with the outside world but not affect the rendering or performance of the component that it's in.

→ Basic syntax:

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

Runs after comp. renders ↑ — ↑ on which side effect relies upon

Two arguments : a function and an array.

callback func. — dependencies array

Miracle
Pg.No :
Date: / /20

Agar array ni value change thai hoi, it will execute the useEffect function again.

→ If we do not provide the dependencies array, it will re-render everytime. This can lead to an infinite loop.

→ What is cleanup function in useEffect()?

Some side effects may need to clean up resources or memory that is not required anymore, avoiding any memory leaks that could slow down your applications.

e.g. Setting up subscription to an external data source. In that scenario it is imp. to perform a cleanup after the effect finishes its execution.

The cleanup function will be called when the component is unmounted.

going to new page / new route

```
useEffect(() => {
    let interval = setInterval(() => setTime(1), 1000)
    return () => {
        clearInterval(interval)
    }
}, [])
```

Miracle
Pg.No :
Date: / /20

Day-6

* What are the rules of hooks?

(1) Only call hooks from a React component function

(2) Only call hooks at the top level.

(3) Can call multiple state or effect hooks inside a component

(4) Make multiple hook calls in the same sequence.

(5) Must call hook before a return statement outside of loops, conditions or nested functions.

* Data fetching using hooks

```
useEffect(() => {
    fetch('https://.....')
      .then((response) => response.json())
      .then((data) => setData(data))
      .catch((error) => console.log(error))
}, [])
```

setData is defined above in a useState hook. So, all the data from API will be stored in that.

## ✱ useReducer hook

→ It is a superpower useState.

→ useState : initialState
useReducer : initialState & reducer function
action object
(has multiple type values)

→ The useReducer hook is best used for more complex data, specifically, arrays and objects.

→ useReducer hook is used to control state of large state-holding objects.

→ e.g.

```
import ....

const reducer = (state, action) => {
  if (action.type === "buy") {
    return {money: state.money - 10}
  }

  if (action.type === "sell") {
    return {money: state.money + 10}
  }

  return state
}
```

```
function App() {

  const initialState = { money: 100}
  const [state, dispatch] = useReducer
                    (reducer, initialState)

  return (
   <><h1> {state.money} </h1>

     <div>
       <button onClick={() => dispatch (
          {type: 'buy'})} >
         Buy
       </button>

      <button onClick={()=> dispatch ({type: 'sell'})}
         Sell
      </button>
     </div>
   </>
```

* useRef hook

→ Allows to directly create a refrence to the DOM element in the functional component.

→ If we change a value in useRef it will not re-render the component.

→ Returns an object

Pg No :
Date : / /20

Day-7

## ✱ JSX, Components and Elements

→ JSX is a Syntax extension to Javascript.

→ React uses JSX to describe UI appearance.

→ Component : Combines markup and business logic

→ Tree of Components :

```
Component
    Comp.
    Comp.
Comp.
```

→ The final web pages React produces is nothing but pure HTML, CSS and JS.

→ JSX :

```
const buttonTitle = 'Submit'
return (
  <button className=" ">
    <span>
      {buttonTitle}
    </span>
  </button>
)
```

Element :

```
{
  type: 'button'
  props: {
    className: ' '
    children: {
      type: 'span',
      children: 'Submit
    }
  }
}
```

→ In element, each node instead of being JSX is a plain object describing a component instance or DOM node and its desired properties.

→ An element is just a way to represent the final HTML output as a plain object.

→ React : Identifies all user defined components
↓
Converts into DOM elements
↓
This result is known as virtual DOM

→ React transforms the JSX into an internal tree of elements, which are just Javascript objects.

✱ Component Composition with children

→ Component Composition
↓ Two main features

↓ Containment

Refers to the fact that some components don't know their children ahead of time. Can also be described as generic boxes, like a Dialog or Alert.

↓ Specialization

Defines component as being "special cases" of other components. For eg. ConfirmationDialog is a special case of Dialog.

Miracle
Pg No :
Date: / /20

Day-8

## ✱ props.children

→ props.children is a special prop, automatically passed to every component, that can be used to render the content included between the opening and closing tags when invoking a component.

→ This components are identified as "Boxes".

→ Example:

```
function Wrapper(props) {
    return <div>{props.children}</div>
}

function App() {
    return (
        <Wrapper>
            <h1> Hello! </h1>
            <p> .... </p>
        </Wrapper>
    )
}
```

So, when we write props.children, we can pass any content but the opening and closing tags. So, it will take <h1>, <p> content and render it between the  of Wrapper component.

Miracle
Pg.No :
Date: / /20

# Types of children

1) String literals

<Wrapper> Hello World! </Wrapper>

→ Rules:

(1) JSX removes whitespaces at the beginning and end of a line, as well as blank lines.

```
<div>   Hello   </div>

<div>
    Hello
</div>
```

↳ Output: Hello

(2)
```
<div>

    Little Lemon
</div>
```

↳ Output: Little Lemon (ignores space)

(3)
```
<div>
    Hello
    World
</div>
```

↳ Output: Hello World (Changes it into single space)

Miracle
Pg No :
Date: / /20

2) JSX elements

```
<Wrapper>
    <Title />
    <Body />
</Wrapper>
```
→ Nested components.

We can also pass different types of children combinations together.

3) Javascript Expressions

```
<Wrap>{`Little Lemon`}</Wrap>
```

4) Functions ✓ (Accepted)

5) Booleans, nulls and undefined are ignored.

```
<div>{undefined}</div>
<div>{true}</div>
<div />
```

They don't render anything.

* Manipulating children dynamically in JSX.

→ Two API's (we will work with these two)

- React.cloneElement
- React.children

Miracle
Pg.No :
Date: / /20

(1) React.cloneElement

→ React top-level API. Used to manipulate and transform elements.

→ Top-level API refers to the way you import these functions from react package.

(1) React global Object

React.cloneElement(...)

(2) As a named Import.

import {cloneElement} from 'react'

cloneElement(...)

→ cloneElement effectively clones and returns a new copy of a provided element.

cloneElement(element, [props])

↑ the element you want to clone

↑ additional props you want to add.

What we can do?

(1) Modify children properties
(2) Add to children properties.
(3) Extend functionality of children

Miracle
Pg No :
Date: / /20

(2) React.children

→ Top-level API, useful for children manipulation.

→ Provides utilities for dealing with the props.children data structure

→ Most imp. method : map function

invokes a func. in every child

React.children.map(children, callback)

of children prop.

↳ Similar to map in React arrays.

Returns a new element

✡ Spread Attributes

→ Spread Operator (...)

When we want to keep the previous data as it is and add new data or modify any data we use spread operator.

e.g
```
const order = {
    id: '1',
    name: '',
    item: 'pizza',
    price: '',
}
```

```
const orderAmen = {
    ... order,
    item: 'burger'
}
```

→ Everything else as it is.

→ Item property changed.
pizza → burger

```
const orderAmen = {
    ... order,        → Everything else as it is.
    item: 'burger'    → Item property changed.
}                       pizza → burger
```

Day-9

## ✱ Cross-cutting concerns in React

→ Permission roles
Handling errors
Logging

They are all required for any application, but are not necessary from business point of view.

This group of functionality is what falls into the category of cross-cutting concerns

→ For eg. an application wants to keep track of like orders list & Newsletter list. They almost have same code (same pattern)

→ So, how to define a logic in single place and share among many components?

⇓

Higher Order Components (HOC)

→ HOC is a function that takes a component and returns a new component.

→ Second method is Render Props.

→ It is a component with a render prop that takes a function that returns a react element and calls that function inside its render logic.

e.g
```
<MealProvider render={data => (
    <p> Ingredients : {data.ingredients} </p>
)}
/>
```

✱ Why React Testing Library ?

→ Why testing ?

- → Discover bugs
- → Ensures software quality
- → Save time and money

→ Practices to keep in mind when writing your tests

- → Avoid including implementation details
- → Work with actual DOM nodes
- → Resemble software usage
- → Maintainability

→ Tools for testing

- → Jest

- Javascript test runner
- Provides access to jsdom
- Good iteration speed
- Powerful features: like mocking modules (more control on how the code executes)

→ React Testing Library

- Set of utilities that let you test React components without relying on their implementation details.
- Fulfils all best testing practices.