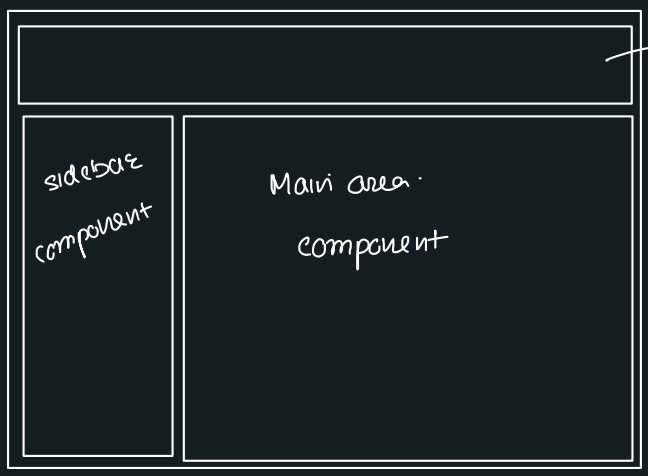
React JS

What is React?

a Javascript library for building User Interfaces → Components

any webpage can be seen as composed of components



Why to think in terms of component?

When we use component based approach we dont build a single big app Rather we build all these components individually and glue them together which then appears as a single application

Advantages of using components:

01) Easy to work in teams

02) Keep code Managble and Maintainable

03) Component Reusability

Why React ?

- UI state becomes difficult to manage with Vanilla Javascript.
- Focus on Business logic and not on development.
- Huge Ecosytem, High performance, Active community.
- Backed by Facebook

React Alternatives

- Angular
- Vue
- Backbone
- jQuery (Not so much)

Two Kinds of Applications

with all these frameworks we can build two types of applications

- Single Page Application
- Multi Page Application

SPA : we only get back one HTML page from server. Content is rendered on client using framework

Entire page is managed by React.

Typically only one ReactDOM.render() call.

Popular way of building webapp these days

MPA: Multiple HTML pages are sent by server on different routes.

although these pages can contain React components along with normal HTML CSS. But the page is not controlled by React.

There is one ReactDOM.render() call per widget

Next Generation JavaScript:-

- Understanding let and const

  Two keywords for creating variables.

  Constant → variables whose value never changes (single assignment)

  let → variables that allows changes in value.

- ARROW FUNCTIONS

```
function HelloWorld(name) {
    console.log(name)
}
```

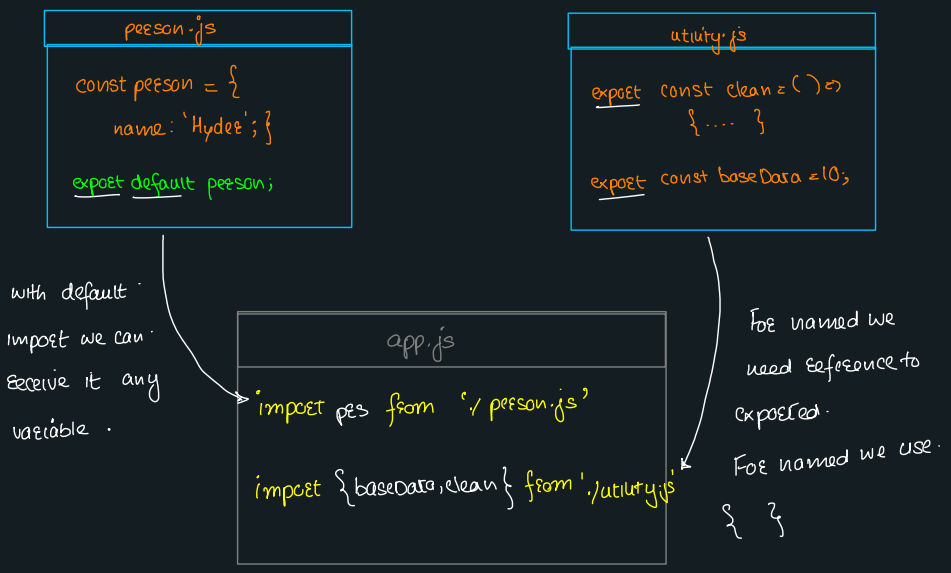
⟶

```
const HelloWorld = name =>
    console.log(name)
```

Exports and Imports (Modules):

There are two types of exports

- Default Exports
- Named Exports



with default export we can choose any name while importing eg.

import pes from './person.js'

→ NOTE: with named exports we strictly have to use exact names as declared in the file. eg

import {clean} from './utility.js'

However we can use Alias as follows

import { clean as cln } from './utility.js'

Classes: are blueprints for objects. Classes contain properties and methods.

```
class Person {
    name: 'Hydee'        → property : value.
    call = () => { ... }   ← method
}
```

NOTE: methods are created only using method name.

USUAGE: `const Hydee = new Person()`

Classes, properties and Methods:

There is a more modern syntax that spares use of constructor function

ES6

```
class XYZ {
  constructor() {
    this.propertyName = 'value'
    ..
  }
  myMethod() { ... }
}
```

ES7.

```
class XYZ {
  propertyName = value;
  myMethod = () => { .... }
}
```

new syntax uses Arrow function. Advantage of using Arrow Function is that it eliminates problems with this keyword.

• Spread and Rest Operators :

actually both spread and rest are same operator. 3 dots "..." It is named differently based on situation it is used.

Spread → used to split up an array or object properties

eg `const newArray = [...oldArray, 1, 2]`

`const newObject = {...oldObj, newprop: 5}`

Rest Operator → used to merge a list of function arguments into an array.

```
function showArgs(...args) {
    console.log(args)
}
```

showArgs (1,2,3,4,5)

→ output : [1,2,3,4,5]

• Destructuring : allows to extract array elements or obj properties and store them in variables

Array Destructuring

```
let [a,b] = [10,20]

// a = 10
// b = 20
```

Object Destructuring

```
let {name} = {name: "Hydee",
              age : 23}
```

variable name must match obj property.

// name = Hydee.

• References and Primitive Types

numbers strings booleans are primitive

objects and Arrays are non primitive

eg let num1 = 10

let num2 = num1.

console.log(num2) // 10

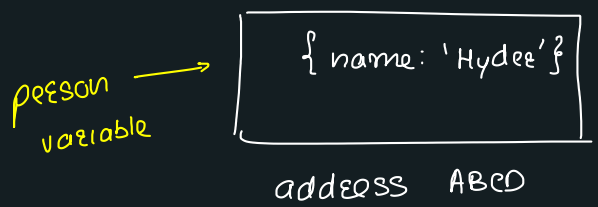
num2 = 5

console.log(num1) // 10

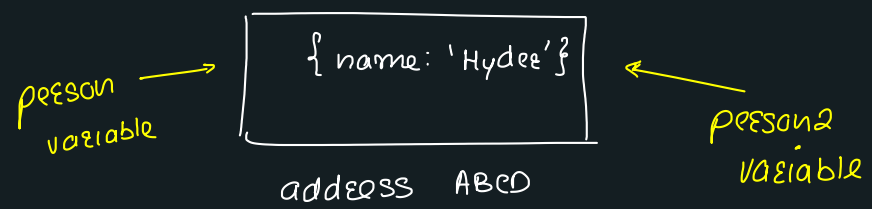
for primitive types values are copied

For non primitive types assignment means referencing

eg const person = { name: "Hydee" }



const person2 = person ; By this assignment person2 will also refer to same memory address



Therefore any change made will reflect by both variables since both of them are pointing to same memory location.

Thus objects should not be copied using assignment but using spread operator.

∴ newobject = { ... oldObject }

• Array Functions

• map → buildin array method that returns a new Array

```
let numbers = [1, 2, 3]
let doubled = numbers.map(num => num * 2);
console.log(doubled)
```

doubled // [2, 4, 6]

Basics : Core React Concepts

Building a Workflow :

- Dependency Management Tool : npm or yarn
- Bundler : Webpack → write Modular code & bundle it together
- Compiler : Babel + presets [change NextGen JS to ES5]
- Development Server

- Using create react app :

It is a tool maintained by Facebook and community that automates entire workflow discussed above.

- Install NodeJS
- npx create-react-app . / npx create-react-app myapp

create files & structure directly in the directory

creates a new directory called myapp and then creates structure.

Understanding React Folder Structure:

public
- → index.html → There is only one HTML file
- → manifest.json

} This folder is finally served by the server at the end of building

src
- contains all react related files
- App.js
- index.js

Component Basics

```
import React, { Component } from 'react'
import './App.css'

class App extends Component {
    render() {
        return (
            <div className="App">
                <h1> Hello World </h1>
            </div>
        )
    }
}

export default App;
```

JSX code (Syntax Sugar) converted to JS automatically

every class based component needs to have atleast a render method. The render method should return either HTML or null.

→ This is exported so that it can be used throughout project

- Understanding JSX:

```
<div className='App'>
    <h1> Hi there </h1>
</div>
```

```
React.createElement ('div', {className:'App'},
React.createElement ('h1', null, 'Hi there'))
```

JSX → compiled to → Normal Javascript.

There are certain restrictions while using JSX eg we cannot use reserved keyword eg class is reserved in Javascript thus we use className.

- It should return a single root element.

- Creating a Functional Component:

```
import React from 'react';                    (1) import react.

const Person = () => {                        (2) component logic
    return (
        <div>
            <h1> Hello World </h1>
        </div>)
}

export default Person;                        (3) export component.
```

- Working with peops : allows us to poss data feom paeent (weapping) element to child (embeded) component.

```
const Peeson = (peops) => {
  eetuen (
  <p> My name is { peops.name } </p>.
  );
}
```

```
<Peeson name = 'Hydee' />
```

NOTE: foe class based components peops is accessed via this.peops

Undeestanding Childeen.

peops object has a speciae peopeety called as childeen

```
<Peeson >
  <h1> Hello Woeld </p> } childeen
</Peeson >
```

Undeestanding and using STATE :

Class has peopeeties. In any component which extends Components we can use speciae peopeety called state as following

```
class App extends Component {
    state = {
      name: 'Hydee', age: 23 }
```

```
Eendee() {
  Eeturn ( <div>
    <p> My name is { this.state.name} </p>
    </div>
  )
}
}
```

accessing state

Note: if state changes it will lead React to Re-Endee entire component.

```
import React from "react";

class Person extends React.Component {
  state = {
    name: "Hyder",
  };

  clickHandler = () => {
    this.setState({ name: "Me" });
  };

  render() {
    return (
      <div>
        <p> My name is {this.state.name}</p>
        <button onClick={this.clickHandler}> Change Text </button>
      </div>
    );
  }
}

export default Person;
```

① event handlee.

event

Eefeeence to handlee

for handling events we only pass eefeeence to handlee function

- Handlee functions aee usually ARROW Functions to avoid this based issues.

- State Managcment in Functional Components using Hooks:

peioe to veesioin 16.8 of React, Closs based components weee only possible way of having state inside React Components.

incase of functional components we use Hooks to manage state.

```
import React , { useState } from 'react';

const Person = props => {

const [personState , setPersonState] = useState( { name : "Hyder" } )
```

what it actually does is that it creates a variable called as personState. Initially personState is initialized to value passed to useState Hook as argument. That is, personState = {name : "Hyder"}

for changing the state we use setPersonState function

```
import React, { useState } from "react";

export default function Person2() {
 let [name, setName] = useState("Hyder");

 const clickHandler = () => {
  setName("Someone");
 };

 return (
  <div>
   <p>{name}</p>
   <button onClick={clickHandler}>Change</button>
  </div>
 );
}
```

① importing React and useState Hook.

② useState Hook — name variable is initialized to value 'Hyder'

③ Event Handler.

→ using setName function we can modify of state name equal to argument passed.

④ handler Response

Stateful vs Stateless Components:

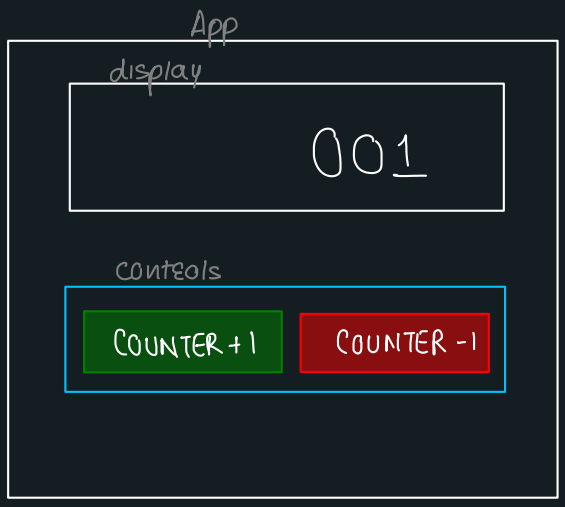
Components either class based or functional which have internal state are called smart components / containers.

While as components with no internal state are called DUMB or presentational components.

We should have more stateless components & few statefull components

• Passing Method References between Components:

Suppose our React app follows below described structure



App component contains state which keeps track of current counter value.

This state is passed to display component as prop.

However counter Handlers for changing the state of current counter value.

needs to be defined in App and reference to these handlers is passed to controls component via props

src > App.js > App > countUpHandler

```jsx
import React, { useState } from "react";
import logo from "./logo.svg";
import "./App.css";

import Display from "./components/display";
import Controls from "./components/controls";

function App() {
  let [count, setCount] = useState(0);

  let countUpHandler = () => {
    setCount(count + 1);
  };

  let countDownHandler = () => {
    setCount(count - 1);
  };
  return (
    <div className='App'>
      <Display count={count} />
      <Controls
        up={() => {
          countUpHandler();
        }}
        down={() => {
          countDownHandler();
        }}
      />
    </div>
  );
}

export default App;
```

① handler fn is defined inside stateful component.

② it is wrapped in an arrow fn and passed as prop

③ event is connected to prop reference.

src > components > display.js > ...

```jsx
import React from "react";

export default function display(props) {
  return (
    <div>
      <input class='display' disabled='true' value={props.count} />
    </div>
  );
}
```

src > components > controls.js > controls

```jsx
import React from "react";

export default function controls(props) {
  return (
    <div className='controls'>
      <button onClick={props.up} className='counter_up'>
        Counter + 1
      </button>
      <button onClick={props.down} className='counter_down'>
        Counter - 1
      </button>
    </div>
  );
}
```

src > components > controls.js > controls

```jsx
import React from "react";

export default function controls(props) {
  return (
    <div className='controls'>
      <button onClick={props.up} className='counter_up'>
        Counter + 1
      </button>
      <button onClick={props.down} className='counter_down'>
        Counter - 1
      </button>

      <input
        onKeyPress={(e) => {
          if (e.key === "Enter") {
            let value = parseInt(e.target.value);
            props.textInc(value);
          }
        }}
      />
    </div>
  );
}
```

```jsx
import Display from "./components/display";
import Controls from "./components/controls";

function App() {
  let [count, setCount] = useState(0);

  let countUpHandler = () => {
    setCount(count + 1);
  };

  let countDownHandler = () => {
    setCount(count - 1);
  };

  let textIncHandler = (value) => {
    setCount(count + value);
  };
  return (
    <div className='App'>
      <Display count={count} />
      <Controls
        up={() => {
          countUpHandler();
        }}
        textInc={(value) => {
          textIncHandler(value);
        }}
        down={() => {
          countDownHandler();
        }}
      />
    </div>
  );
}

export default App;
```

• Adding Styling

There are two ways to style components.

• Global Styling with CSS files. All these CSS files are globally scoped. However we do need to import it in JS so that webpack includes it in the bundle.

• Inline Styles:

```
1   import React from "react";
2
3   export default function controls(props) {
4     const styles = {
5       counterUp: {
6         backgroundColor: "green",
7         color: "white",
8       },
9
10      counterDown: {
11        backgroundColor: "red",
12        color: "white",
13      },
14    };
15
16    return (
17      <div className='controls'>
18        <button
19
20          style={styles.counterUp}
21
22          onClick={props.up}
23          className='counter_up'>
24          Counter + 1
25        </button>
26        <button
27
28          style={styles.counterDown}
29
30          onClick={props.down}
31          className='counter_down'>
32          Counter - 1
33        </button>
34
35        <input
36          onKeyPress={(e) => {
37            if (e.key === "Enter") {
```

① Inline styles

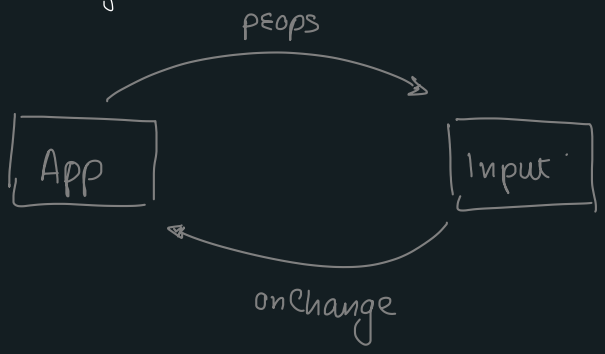
② using Inline styles.

JSX provides a special property called as style. This can be used to apply above defined styles to an element in JSX

• Two way binding

whenever input changes onChange event will free which intern will change state of App. Since state of App component changes, peops passed to Input component will also change.

```
function App ( ) {
    let [ name , setName ] = useState ('Hydee')
    const handleChange = ( newValue ) => {
        setName ( newValue )
    }
    return (
      <div>
        < Input value = { name } handleChange = { handleChange }/>
      </div>
    )
}
```




```
function Input (peops) {
    return (
    <input value = { peops.value } onChange = {
(e) => { peops.handleChange (e.target.value) }.
    )
}
```

• Rendering Content Conditionally

```
class App extends React.Component {
    state = { show : false }
    rendee ( ) {
        return (
          <div>
```

```
{ this.state.show ? <p> Hello World </p> : null }
    <p> Does work </p>
    </div>
  )
 }
}
```

{ } contains JS expressions which are evaluated to a single value. However we cannot use if else directly inside { }. In this case we are using Ternary operator.

- **Better Approach to conditionally rendering**

For large and more complicated apps this approach is not very optimal.

We can create a function that is responsible for rendering that part and call that function inside return of JSX. eg.

```
function Person (props) {
    let [show, setShow] = useState (false)
→   function showPerson ( ) {
        if (show) return ( <p> Hello World </p> )
        else return null
    }
    return ( <div>
        { showPerson ( ) } → calling function.
        </div>
    )
}
```

Method responsible for conditional rendering

- # Rendering a list of elements using map:

we can render a list/array using map function as shown below:

src > components > display.js > Display > renderList > people.map() callback

```
1   import React, { useState } from "react";
2
3   export default function Display(props) {
4     let [show, setShow] = useState(true);
5     let [people, setPeople] = useState([
6       {
7         name: "Hyder",
8       },
9       {
10        name: "Jan",
11      },
12      { name: "Manan" },
13    ]);
14
15    let renderList = () => {
16      if (show) {
17        return people.map((person) => {
18          return (
19            <div>
20              <p> {person.name} </p>
21            </div>
22          );
23        });
24      } else return null;
25    };
26
27    return (
28      <div>
29        {renderList()}
30        <input class='display' disabled='true' value={props.count} />
31      </div>
32    );
33  }
34
```

people holds an array of objects

renderList method is responsible for dynamically generating JSX for list using map fn.

JSX returned for each element of array.

• Updating State Immutably

We should never set state directly but always use functions such as • setState • function declared during state Hook to set state only. The reason for this restriction is that React uses them to re-render the components on state change.

If we directly set state there will be no component re-rendering

also we should copy state using spread operator. Since state is a JS object assignment will only create another reference to state object and not copy it.

• List and Keys

for lists there is a special property called key. Key for each element is unique. It uses key to allow react to efficiently modify virtual DOM rather than re-rendering entire list which for very large lists can be very inefficient.

Styling React Components

Most basic limitation of using inline style is that we cannot use Pseudo classes such as Hover, after etc. Now we can use it via CSS file but then it would be globally scoped.

Radium: is a 3rd party package for React that allows us to overcome restrictions of inline styles. Allows us to use Pseudo selectors, media queries

- npm i --save radium → installs Radium package
- import Radium from 'radium'
- wrap component in Radium( )

src > components > controls.js > controls

```
1   import React from "react";
2
3   import Radium from "radium";
4
5   function controls(props) {
6     const styles = {
7       counterUp: {
8         backgroundColor: "green",
9         color: "white",
10        tranistion: "transform all .2s",
11        ":hover": {
12          transform: "scale(0.8)",
13        },
14      },
15
16      counterDown: {
17        backgroundColor: "red",
18        color: "white",
19      },
20    };
21
22    return (
23      <div className='controls'>
24        <button
25          style={styles.counterUp}
26          onClick={props.up}
27          className='counter_up'>
28          Counter + 1
29        </button>
30        <button
31          style={styles.counterDown}
32          onClick={props.down}
33          className='counter_down'>
34          Counter - 1
35        </button>
36
37        <input
38          onKeyPress={(e) => {
39            if (e.key === "Enter") {
40              let value = parseInt(e.target.value);
41              props.textInc(value);
42            }
43          }}
44        />
45      </div>
46    );
47  }
48
49  export default Radium(controls);
50
```

(1) import Radium

This shows we need to wrap pseudo selectors in quotation marks

This is equivalent to:

```
.counterUp:hover {
    transform: scale(0.8);
}
```

(2) export component wrapped in Radium

Introducing Styled Components:

Styled components library for styling components with ease

- npm i --save styled-components
- import styled from 'styled-components';

const Button = styled.button``

↳ regular JS feature called tagged templates

understanding styled.button`` syntax:

Here styled is an object and button is a method. Instead of having parenthesis we are having backtics in which we can pass text which is passed into button method.

This styled object that we are importing has a method for each HTML element eg

styled.button``  styled.p``

styled.h1`` ... etc.

Code Example of Styled Components :

src > components > counter_styled.js > Counter_Styled > IncreaseBtn

```
1   import React, { useState } from "react";
2
3   import styled from "styled-components";
4
5   function Counter_Styled() {
6     let [count, setCount] = useState(0);
7
8     let Div = styled.div`
9       margin: auto auto;
10      margin-top: 60px;
11    `;
12
13    let Counter_Display = styled.div`
14      font-size: 20px;
15      color: gray;
16    `;
17
18    let IncreaseBtn = styled.button`
19      background-color: green;
20      padding: 20px;
21      color: white;
22
23      :hover {
24        transform: scale(0.8);
25      }
26    `;
27    return (
28      <Div>
29        <Counter_Display>{count}</Counter_Display>
30
31        <div>
32          <IncreaseBtn
33            onClick={(e) => {
34              setCount(count + 1);
35            }}>
36            Increase Counter
37          </IncreaseBtn>
38        </div>
39      </Div>
40    );
41  }
42
43  export default Counter_Styled;
44
```

① importing styled components

② Div is a div with all these styling preattached.

② Counter-Display is a div which is already prestyled.

Pseudo selector.

@

ⓑ

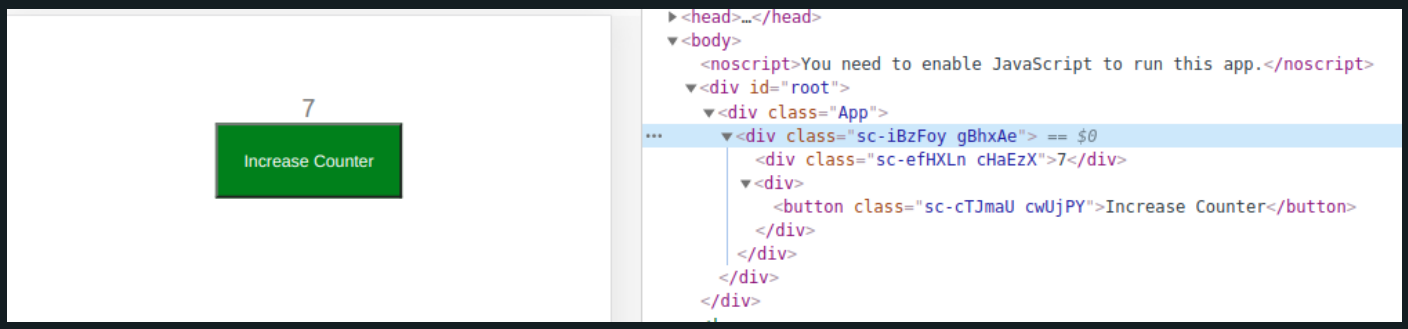
ⓒ

instead of using JSX we use variables which are generated by styled components as marked by @ , ⓑ , ⓒ , ⓓ

How Styled components work internally?

Using previous counter code we wrote as styled component we will try to understand how Styled components works internally

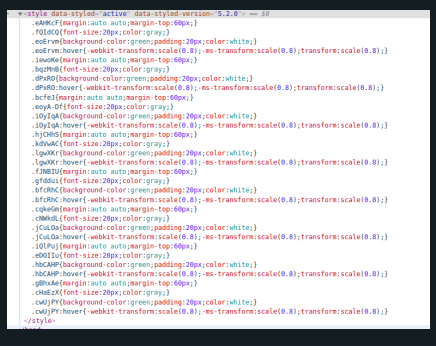
If we inspect our react app using developer tools, we will notice elements with strange class names attached to them



These classes are generated by styled components library.

These classes are defined in the head of HTML.

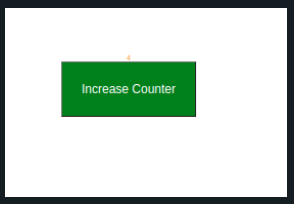
Thus style components takes our CSS that we pass in backtics places in <style> tag in the head section with a unique name. It just adds elements with appropriate classes.



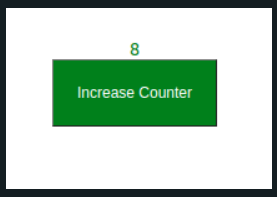
→ styles created by styled components library inside head section of HTML

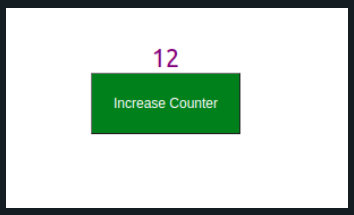
Dynamic Styles with Styled Components:

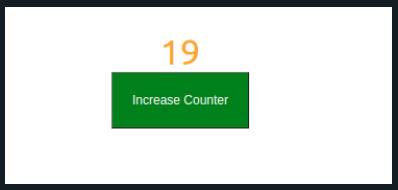
We use props to control dynamic styles. We pass props to component. This prop is then accessible inside backticks that we attach to that particular component.



Smaller the value of count smaller is the size and color also differs







src > components > counter_styled.js > Counter_Styled

```
1  import React, { useState } from "react";
2
3  import styled from "styled-components";
4
5  function Counter_Styled() {
6    let [count, setCount] = useState(0);
7
8    let Div = styled.div`
9      margin: auto auto;
10     margin-top: 60px;
11   `;
12
13   let Counter_Display = styled.div`
14     font-size: ${(props) => {
15       return props.count * 2;
16     }}px;
17
18     color: ${(props) => {
19       if (props.count > 5 && props.count < 10) {
20         return "green";
21       } else if (props.count > 10 && props.count < 15) {
22         return "purple";
23       } else return "orange";
24     }};
25   `;
26
27   let IncreaseBtn = styled.button`
28     background-color: green;
29     padding: 20px;
30     color: white;
31
32     :hover {
33       transform: scale(0.8);
34     }
35   `;
36   return (
37     <Div>
38       <Counter_Display count={count}>{count}</Counter_Display>
39
40       <div>
41         <IncreaseBtn
42           onClick={(e) => {
43             setCount(count + 1);
44           }}>
45           Increase Counter
46         </IncreaseBtn>
47       </div>
48     </Div>
49   );
50 }
51
52 export default Counter_Styled;
53
```

→ initializing count state.

→ dynamically assigning font size based on count value which is passed via props to component.

→ similarly for dynamically assigning color.

passing count as prop to this styled component value of this props determines color and font-size

SYNTAX

property : ${ (props) => {

logic ... return value ; }.

}

CSS Modules :

- CSS scoped to file or components . Rather than bloating JS with CSS .

To use CSS Modules there is a need to tweak build configuration we need .

We need to eject from react by npm run eject . This will expose all the underlying configurations of webpack present in config folder .

- we need to make changes in two files

webpack.config.dev.js

webpack.config.prod.js

```
{ test : /\.css$/,
  use : [
    require.resolve ('style-loader'),
    { loader : require.resolve ('css-loader'),
    options : {
          importLoaders : 1,
          modules : true,
          localIdentName : '[name]__[local]__[hash:base64:5]'
        },
    },
```

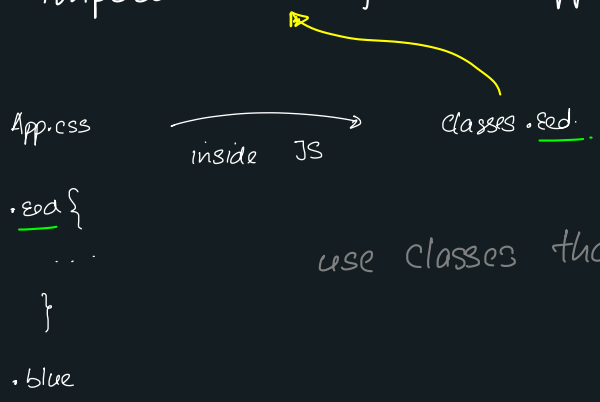
enables modules

we need to add these two lines

provides a unique name to classes in DOM

Now it will also us to import from CSS in another manner.

import classes from './App.css'



Thus CSS Modules allow us to use classes that are scoped to component level only.

Now there are different ways of including CSS in React

- Radium
- Styled Components
- CSS Modules

NOTE: for React scripts greater than 2.0 we donot need to eject for using modular CSS. Rather we can use directly by changing syntax.

① Naming CSS files: XYZ.module.css

② Importing CSS in JS: import classes from './XYZ.module.css'

This allows us to unlock CSS modules without ejecting

- Using Global Styles with CSS Modules:

We can define a global CSS class (means it will not be transformed) by using the following Syntax

```
:global .Post { ... }
```

This makes it global in nature.

we can use a global style directly in any component as `<Component className="Post" />`

Example using CSS Modules without ejection:

App.js | display.module.css | Display.js

src > components > css_module_counter > Display.js > Display

```
1  import React from "react";
2
3  import styles from "./display.module.css";
4
5  export default function Display({ count }) {
6    console.log(styles);
7    return (
8      <div className={styles.display}>
9        <input className={styles.display_input} value={count} disabled />
10     </div>
11   );
12 }
13
```

① importing styles

② using styles

App.js | display.module.css | Display.js

src > components > css_module_counter > display.module.css > .display

```
1  .display {
2    padding: 30px;
3    background-color: aquamarine;
4    text-align: center;
5  }
6
7  .display_input {
8    font-size: 25px;
9    padding: 10px;
10   margin: auto auto;
11   text-align: center;
12 }
13
```

③ writing Normal CSS saving as name.module.css

Debugging React Applications:

- **Understanding Error Messages:**

React shows errors in console by showing line number where error has occured. These are syntactical errors which are easier to debug. However errors also do occur due to incorrect logic and are called as Logical Errors.

- **Finding logical Errors using dev tools & Source Maps:-**

sources → debugger breakpoints

- **React Developer Tools:-** 3rd party extension available for Chrome. Downloadable from chrome web store.

- **Using Error Boundaries**

ErrorBoundaries are higher order components with sole job of handling errors. ErrorBoundary wraps components inside it.

ErrorBoundaries should be used only in cases where we know code can fail. This is used to show custom error messages.

- Implementing Error Boundaries :

01) Create a new folder 'ErrorBoundaries' inside src folder.

02) Create a new ErrorBoundary.js component as following

src > ErrorBoundary > ErrorBoundary.js > ...

```
import React, { Component } from "react";

class ErrorBoundary extends Component {
  state = {
    hasError: false,
    errorMessage: "",
  };

  componentDidCatch(error, info) {
    this.setState({ hasError: true, errorMessage: error });
  }

  render() {
    if (this.state.hasError) {
      return <h1>{this.state.errorMessage}</h1>;
    } else return this.props.children;
  }
}

export default ErrorBoundary;
```

→ catches the error from all the descendants of children components

→ returns errorMessage.

→ else returns all the children components that are wrapped by ErrorBoundary Component.

src > App.js > ...

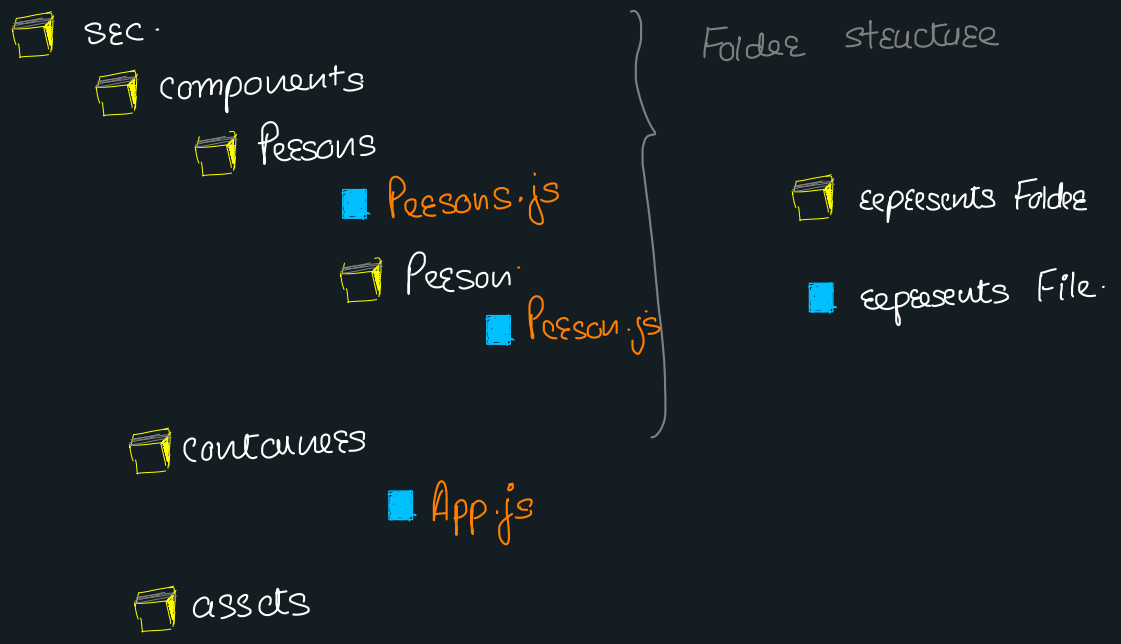
```
import React, { useState } from "react";

import Display from "./components/css_module_counter/Display";

import ErrorBoundary from "./ErrorBoundary/ErrorBoundary";

export default function App() {
  let [count, setCount] = useState(0);

  setTimeout(() => {
    setCount(count + 1);
  }, 1000);

  function renderDisplay() {
    if (count > 10) {
      throw new Error("Count Greater than 40");
    } else return <Display count={count} />;
  }

  return <ErrorBoundary>{renderDisplay()}</ErrorBoundary>;
}
```

- Deeper dive into React Internals and Components:-

→ Components Deep Dive

Project Structure:-

Usually components which are responsible for state Management should not deal with too much JSX. It means for stateful components render method should be clean.



Comparing Stateless and Stateful Components

Stateful component is one which manages state.

Historically class based component referred to stateful while as functional component meant dumb or presentational components.

Generally there should be alot of dumb or presentational components. If we have fewer and fixed number of stateful components then understanding flow of data becomes easy.

This also allows us to reuse presentational components throughout project without any interfaces.

- Class based vs Functional Components

→ class based components: are simple javascript classes that extends React.Component class. Historically these components were only way to manage state in a component.

- Access to State: class based components have direct access to state. Sets state using setState()

- **Lifecycle Hooks:** have access to lifecycle Hooks by default.
- **Access State and Props:** Class based component uses this Keyword because incase of class based components STATE and PROPS are properties of Class.

- **Functional Components:** are Javascript functions that receive props as an argument by default.
  - **Access to state:** is via useState Hook.
  - **Lifecycle Hooks:** are still not perfectly supported.
  - **Access to State and Props:** is via useState() hook and props is directly accessible as an argument.

When to use what: although the diff b/w class based & functional components is narrowing day by day. But still we can choose based on following

- using a version of React which doesnot supports Hooks we will have no choice but to use class based components

- if lifecycle Hooks are required we still need to use class based.

Understanding Component Lifecycle:

First takeaway is that it is only available in class based components

Functional components have only an equivalent of lifecycle available using Hooks.

Class based lifecycle Methods: Any React class based component will have following methods available. React automatically executes these methods at certain point of time

- constructor()
- getDerivedStateFromProps()
- shouldComponentUpdate()
- getSnapShotBeforeUpdate()
- componentDidUpdate()
- componentDidCatch()
- componentDidMount()
- componentWillUnmount()
- render()

Component Lifecycle - Creation :-

When a React Component is created then first of all constructor() method is executed.

1. **constructor(props)**

actually constructor is not React lifecycle Hook but simple JS.

Incase if we donot mention constructor method. It is called automatically by default. However if we explicitly mention constructor call then we also need to call super(props) as first statement of constructor.

- Constructor is to be used for initialization eg setting up state.
- However it should not be used for AJAX calls or localStorage etc. called SIDE AFFECTS.

2. **getDerivedStateFromProps(props, state)**

runs after constructor.

USECASE: whenever props change, we can sync state to them.

③ render() render method should be used to prepare & structure JSX code.

④ Render Child Components renders all child components and once all there lifecycle Hooks finish componentDidMount executes

⑤ componentDidMount() is a very important lifecycle Hook can be used for SIDE EFFECTS eg reaching out to web, causing HTTP server.

NOTE: state should not be updated in this lifecycle Method because it will cause re-rendering.

However state can be set asynchronously. When response is received

• Component Lifecycle - Update (Props Change)

`getDerivedStateFromProps(props, state)` → syncs state to props

`shouldComponentUpdate(nextProps, nextState)` → may cancel updating process
decides whether to continue update or not.

`render()` → prepares & structures JSX

`updateChildComponentProps`

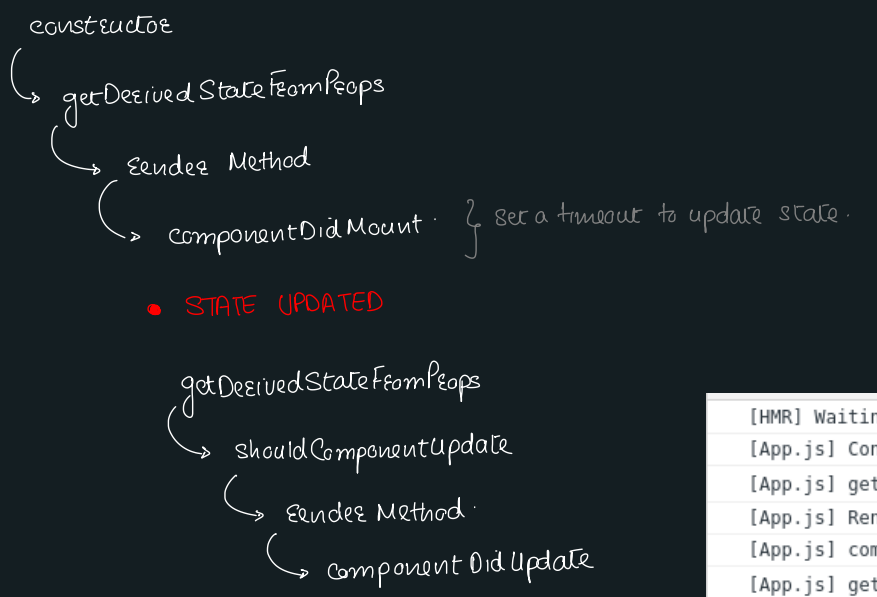
`getSnapshotBeforeUpdate(prevProps, prevState)` → last minute DOM operations

`componentDidUpdate()` → used to perform side effects
eg N/W requests, local Storage Access etc.

Note: we should not try to set state in componentDidMount directly because it will lead to re-rendering.

However state can be set Asynchronously.

Component Update Lifecycle - State Changes




```
[HMR] Waiting for update signal from WDS...
[App.js] Constructor Running!
[App.js] getDerievedStateFromProps Running! ▶ {count: 0}
[App.js] Render Method Running!
[App.js] componentDidMount
[App.js] getDerievedStateFromProps Running! ▶ {count: 1}
[App.js] shouldComponentUpdate ▶ {count: 1}
[App.js] Render Method Running!
[App.js] ComponentDidUpdate
>
```

index.js | App.js

src > containers > App.js > ...

```jsx
import React, { Component } from "react";

class App extends Component {
  constructor(props) {
    super(props);

    console.log("[App.js] Constructor Running!");

    this.state = {
      count: 0,
    };
  }

  static getDerivedStateFromProps(props, state) {
    console.log("[App.js] getDerievedStateFromProps Running!", state);

    return null;
  }

  componentDidMount() {
    console.log("[App.js] componentDidMount");

    setTimeout(() => {
      this.setState({ count: this.state.count + 1 });
    }, 3000);
  }

  componentDidUpdate() {
    console.log("[App.js] ComponentDidUpdate");
  }

  shouldComponentUpdate(nextProps, nextState) {
    console.log("[App.js] shouldComponentUpdate", nextState);
    return true;
  }

  render() {
    console.log("[App.js] Render Method Running!");

    return <div>Hello from App.JS - Count {this.state.count} </div>;
  }
}

export default App;
```

• Lifecycle Methods in Functional Components

Now for functional components we dont have access to lifecycle methods But we use an equivalent Hooks for the same.

React Hooks is supported from React 16.0.8

There are different React Hooks that exist eg useState, useEffect.

• useEffect React Hook

is second most important React Hook after useState.

useEffect combines the functionality or use cases of all lifecycle methods in a single React Hook

• Working of UseEffect React Hook

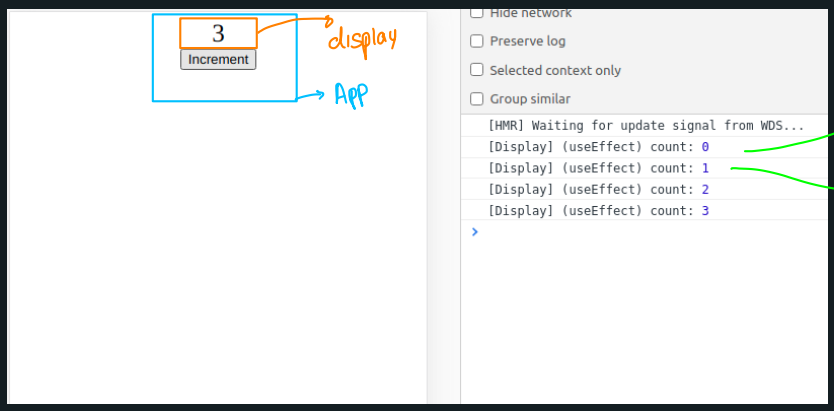
- useEffect Hook takes in a function
- This function runs or is executed everytime component is rendered.

Thus it means useEffect involves → ComponentDidMount
first rendered cycle.

→ ComponentDidUpdate
everytime component renders due to update

lifecycle methods.

Example on useEffect :



Represents the first time display component is rendered.

everytime state of App component is updated, display component will re-render thus useEffect Hook will execute

Important

• Controlling useEffect Behaviour :

As we have already mentioned that useEffect is combination of certain lifecycle methods. eg lifecycle Methods such as componentDidMount & componentDidUpdate.

As default behaviour useEffect will execute everytime component is rendered.

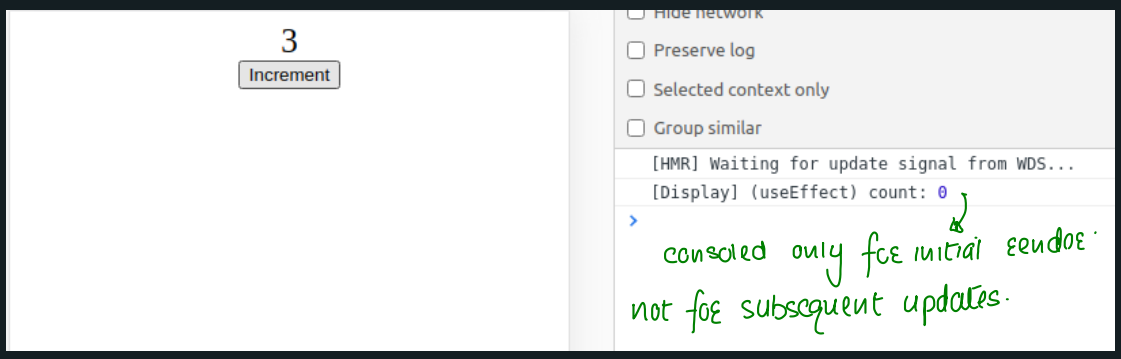
To control the behaviour of useEffect it has a dependency list in the form of an array.

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

callback function — dependency list

- if dependency list is passed as an empty array, callback function will execute only for one time when component is initially rendered. Thus it will be equal to componentDidMount.

useEffect(() => { }, [ ]) = componentDidMount.

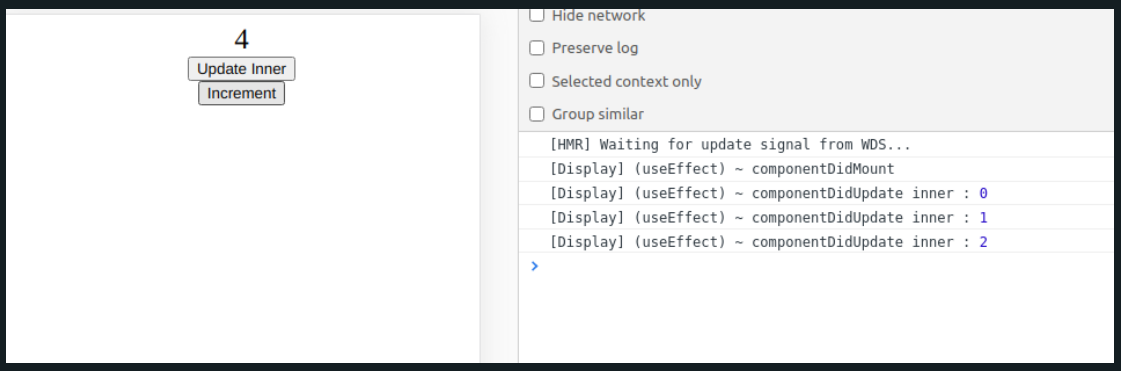


src > components > Display.js > Display

```
1   import React, { useEffect } from "react";
2
3   import styles from "./Display.module.css";
4
5   export default function Display(props) {
6     useEffect(() => {
7       console.log("[Display] (useEffect)", "count:", props.count);
8     }, []);
9     return (
10      <div>
11        <div className={styles.display}> {props.count} </div>
12      </div>
13    );
14  }
15
```

- if dependency list contains some vaerables

useeffect will act as ComponentDidUpdate lifecycle method



src > components > Display.js > ...

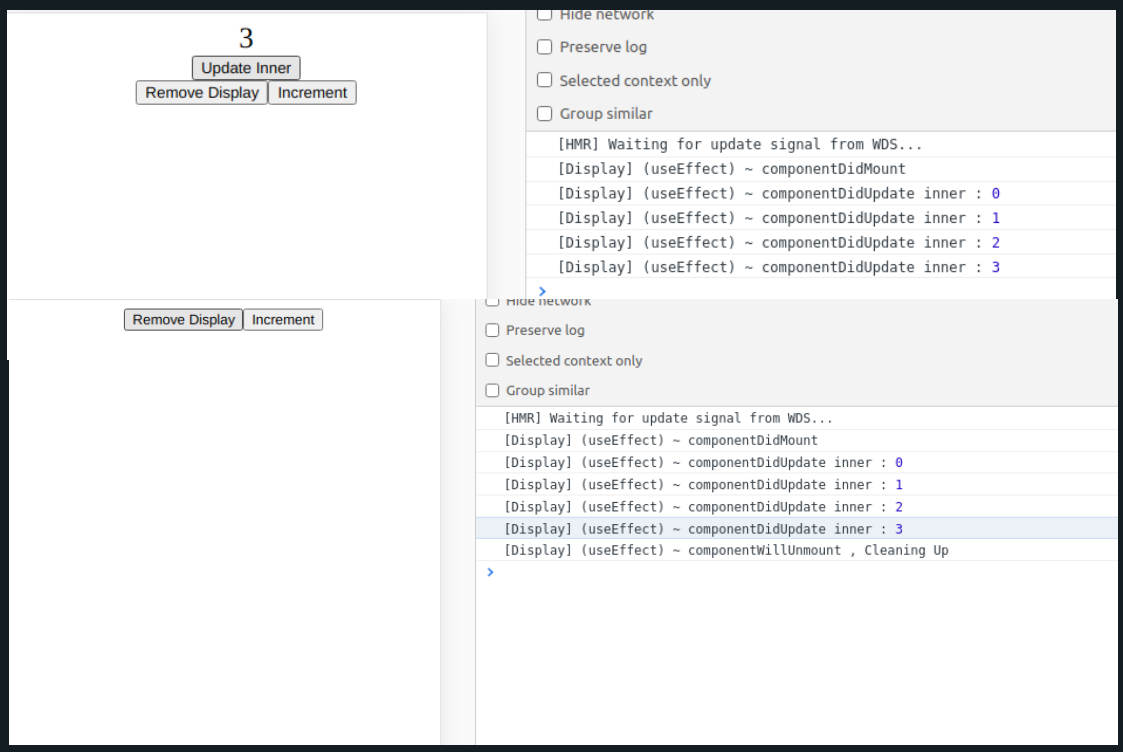
```jsx
import React, { useEffect, useState } from "react";

import styles from "./Display.module.css";

export default function Display(props) {
  let [inner, setInner] = useState(0);

  /* ComponentDidMount */

  useEffect(() => {
    console.log("[Display] (useEffect) ~ componentDidMount");
  }, []);

  useEffect(() => {
    console.log("[Display] (useEffect) ~ componentDidUpdate", "inner :", inner);
  }, [inner]);

  return (
    <div>
      <div className={styles.display}> {props.count} </div>
      <button
        onClick={() => {
          setInner(inner + 1);
        }}>
        Update Inner
      </button>
    </div>
  );
}
```

- Cleaning up with Lifecycle Hooks and useEffect :

Before a component is to be removed, Component class provides us with a lifecycle method known as componentWillUnmount.

This method is used for cleanup work such as disassociating any eventlisteners.

incase of functional components this can be obtained by returning a function inside call back function of useEffect.



src > components > Display.js > Display > useEffect() callback

```jsx
import React, { useEffect, useState } from "react";

import styles from "./Display.module.css";

export default function Display(props) {
  let [inner, setInner] = useState(0);

  /* ComponentDidMount */

  useEffect(() => {
    console.log("[Display] (useEffect) ~ componentDidMount");
    return () => {
      console.log("[Display] (useEffect) ~ componentWillUnmount , Cleaning Up");
    };
  }, []);

  useEffect(() => {
    console.log("[Display] (useEffect) ~ componentDidUpdate", "inner :", inner);
  }, [inner]);

  return (
    <div>
      <div className={styles.display}> {props.count} </div>
      <button
        onClick={() => {
          setInner(inner + 1);
        }}>
        Update Inner
      </button>
    </div>
  );
}
```

empty dependency list

Runs as cleanup function ≈ componentWillUnmount

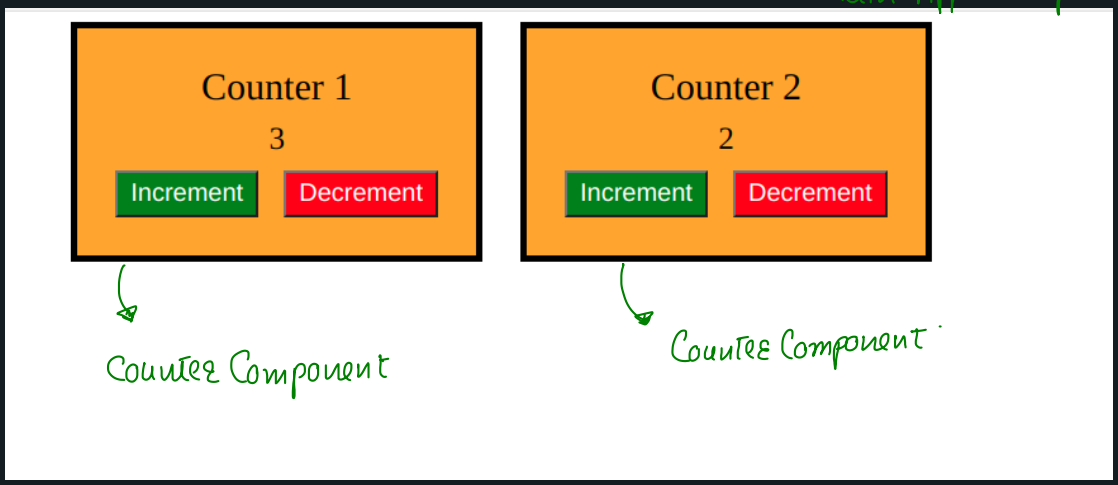
Note for this to act as componentWillUnmount, the dependency list needs to be an empty array.

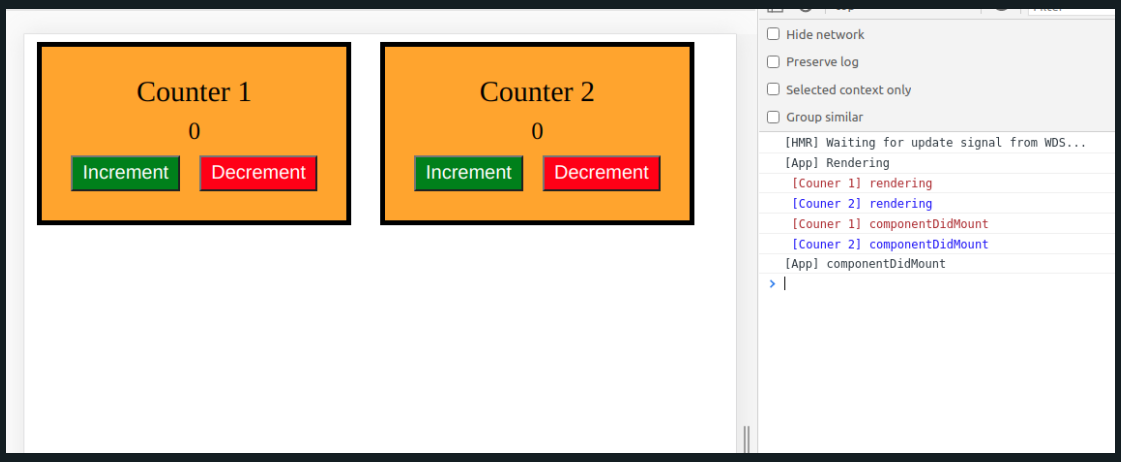
- shouldComponentUpdate for classbased component.

OPTIMIZATION

in react whenever state of parent component changes, all the child components are re-rendered atleast in React Virtual DOM. This is how react works under the Hood. However this behaviour can be optimized by using shouldComponentUpdate Lifecycle Hook.

Let us consider an example: Suppose we have an App component which has 2 child components (counters). The count value is stored by App component. Whenever state of component is updated both child components are re-rendered.





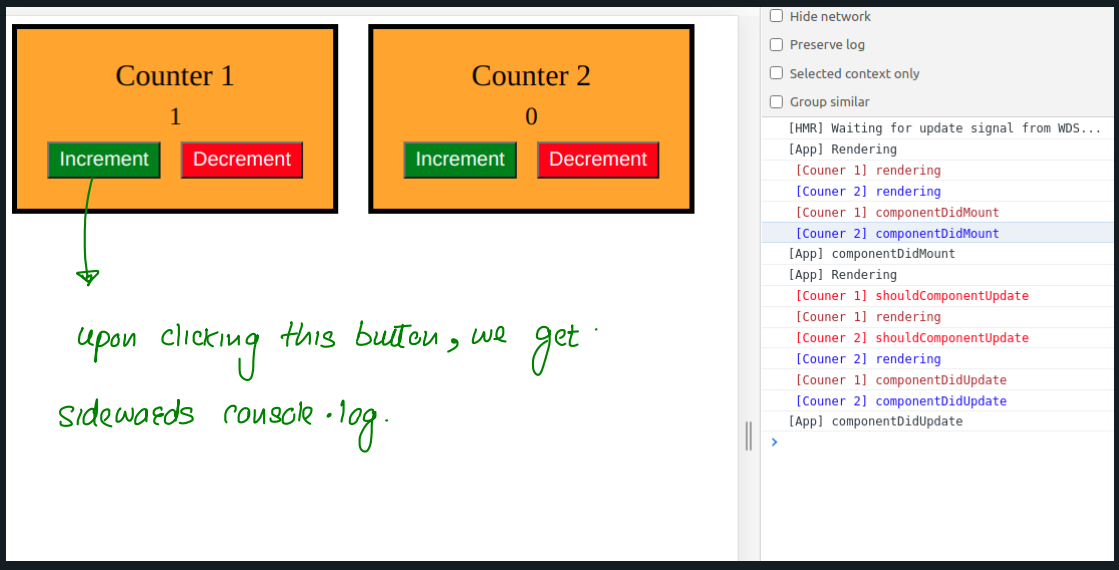
App.module.css | Counter.js | App.js | Counter.module.css

src > containers > App.js > App > componentDidUpdate

```jsx
import React, { useState } from "react";

import Counter from "../components/Counter";

export default class App extends React.Component {
  state = {
    count1: 0,
    count2: 0,
  };

  incrementHandler = (number) => {
    switch (number) {
      case 1:
        this.setState({ count1: this.state.count1 + 1 });
        break;

      case 2:
        this.setState({ count2: this.state.count2 + 1 });
        break;

      default:
        return;
    }
  };

  componentDidMount() {
    console.log("[App] componentDidMount");
  }

  componentDidUpdate() {
    console.log("[App] componentDidUpdate");
  }

  render() {
    console.log("[App] Rendering");
    return (
      <div className='main_wrapper'>
        <Counter
          count={this.state.count1}
          incHandler={() => {
            this.incrementHandler(1);
          }}
          name={1}
        />

        <Counter
          count={this.state.count2}
          incHandler={() => {
            this.incrementHandler(2);
          }}
          name={2}
        />
      </div>
    );
  }
}
```

src > components > Counter.js > Counter > constructor

```jsx
import React, { Component } from "react";

import styles from "./Counter.module.css";

export default class Counter extends Component {
  constructor(props) {
    super(props);

    this.state = {};

    switch (this.props.name) {
      case 1:
        this.state.color = "color:brown";
        break;

      case 2:
        this.state.color = "color:blue";
        break;

      default:
        return;
    }
  }

  componentDidMount() {
    console.log(
      `%c [Couner ${this.props.name}] componentDidMount`,
      this.state.color
    );
  }

  componentDidUpdate() {
    console.log(
      `%c [Couner ${this.props.name}] componentDidUpdate`,
      this.state.color
    );
  }

  shouldComponentUpdate() {
    console.log(
      `%c [Couner ${this.props.name}] shouldComponentUpdate`,
      "color:red"
    );

    return true;
  }

  render() {
    console.log(`%c [Couner ${this.props.name}] rendering`, this.state.color);
    return (
      <div className={styles.wrapper}>
        <div className={styles.name}>Counter {this.props.name}</div>
        <div className={styles.display}>{this.props.count}</div>
        <div className={styles.controls}>
          <button className={styles.green} onClick={this.props.incHandler}>
            Increment
          </button>
          <button className={styles.red}>Decrement</button>
        </div>
      </div>
    );
  }
}
```



as it is clearly noticable that by only incrementing count of a single counter both counters are re-rendered because App component is re-rendered. Clearly this is inefficient.

using shouldComponentUpdate for Optimization:

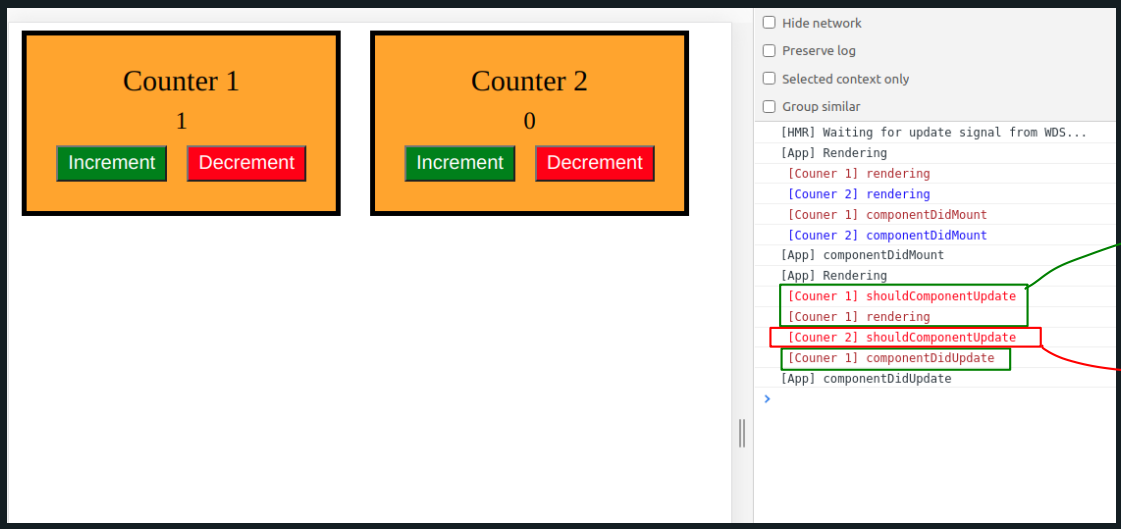
in this lifecycle method we can check if component needs to be rendered. In this particular case we can check that for a counter component prop value is different than existing prop value. If prop values are different, we will update component otherwise we will not update.

App.module.css | Counter.js | App.js | Counter.module.css

src > components > Counter.js > Counter > shouldComponentUpdate

```
30    }
31
32    componentDidUpdate() {
33      console.log(
34        `%c [Couner ${this.props.name}] componentDidUpdate`,
35        this.state.color
36      );
37    }
38
39    shouldComponentUpdate(nextProps, nextState) {
40      console.log(
41        `%c [Couner ${this.props.name}] shouldComponentUpdate`,
42        "color:red"
43      );
44
45      if (nextProps.count !== this.props.count) return true;
46      else return false;
47    }
48
49    render() {
50      console.log(`%c [Couner ${this.props.name}] rendering`, this.state.color);
51      return (
52        <div className={styles.wrapper}>
53          <div className={styles.name}>Counter {this.props.name}</div>
54          <div className={styles.display}>{this.props.count}</div>
55          <div className={styles.controls}>
56            <button className={styles.green} onClick={this.props.incHandler}>
57              Increment
58            </button>
59            <button className={styles.red}>Decrement</button>
60          </div>
61        </div>
62      );
63    }
64  }
65
```

Here we check if component needs to be re-rendered



Counter 1 is re-rendered.

Counter 2 does not re-render because shouldComponentUpdate lifecycle method returns false.

• Optimizing Functional Components using React.memo() :

For functional Components that do not need to change everytime parent component renders we can use React.memo

React.memo → stands for memorization returns snapshot of component.

To use React.memo we need to use it as Higher order component eg:

```
function X42 () {
  return ( ..... )
}
```

```
export default React.memo( X42 )
```

When should we Optimize :

Now when should we consider using these optimizations. We need to understand, that these also come at some cost. If we have parent component that directly holds data of child component.

then checking each time state changes will only add extra burden of execution.

• Pure Components instead of shouldComponentUpdate.

PureComponent is simply a normal component which comes with a preconfigured life Cycle Method that checks for equivalence of all the props.

Thus in situation where shouldComponentUpdate needs to check for equivality of all props it is better to use PureComponent

```
class X42 extends PureComponent { ..... }
```

How React updates the DOM

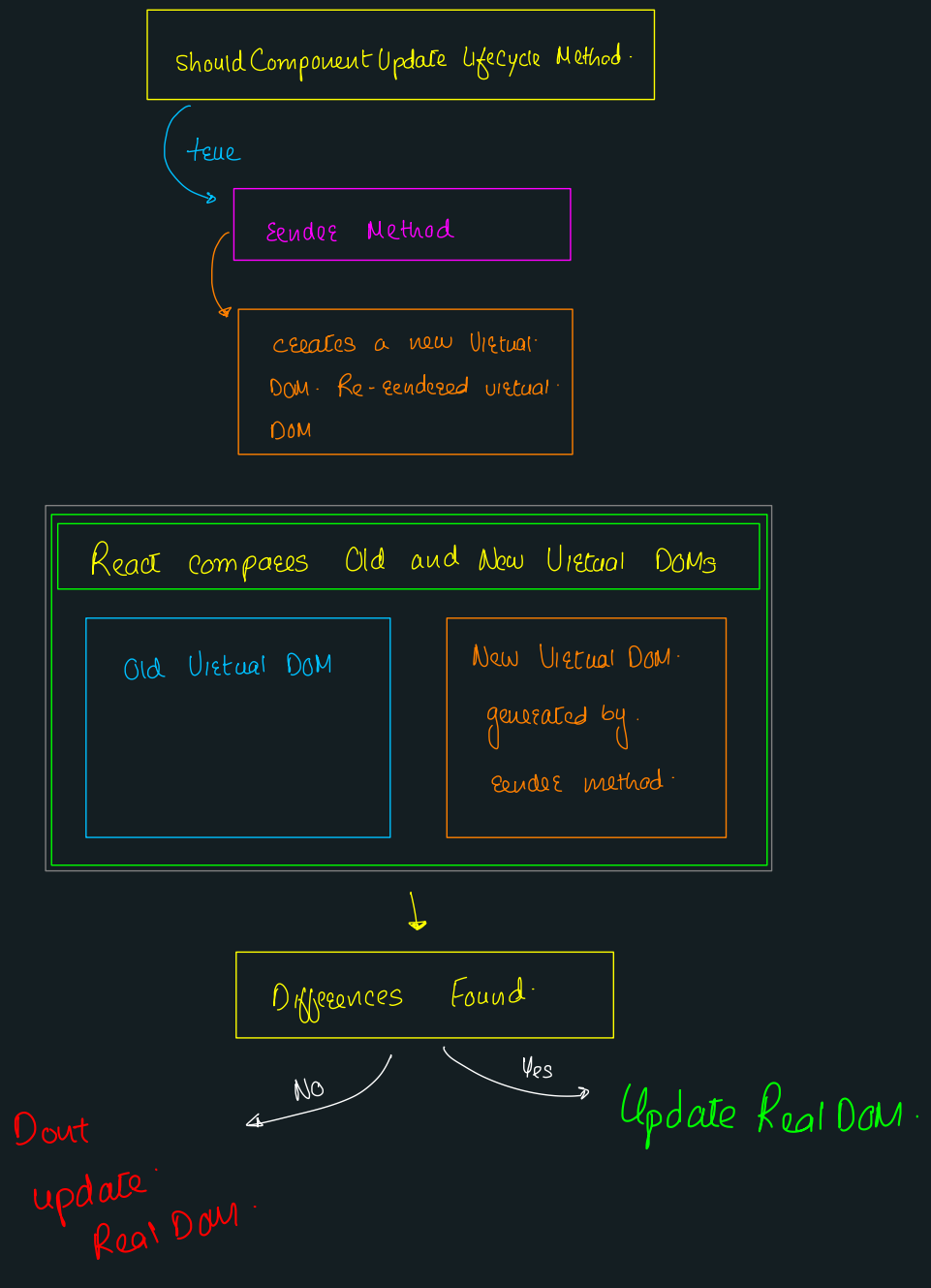
Whenever shouldComponentUpdate lifecycle method returns true. render method is executed. Render method generates virtual DOM. It must be noted that render method does not affect the real DOM directly.

Once new virtual DOM is created, React compares it with old virtual DOM to figure out any differences. If no differences are found, real DOM is left untouched.

If differences are noticed, real DOM is updated accordingly based on the differences and not entirely.

Virtual DOM is pretty fast as compared to real DOM mutations. Thus React tries best to limit interactions with React DOM.

React DOM Update Flow Chart :-



- Rendering JSX Elements :-

Now in React we can return JSX with a single root element. This root element can contain as many as nested child elements. However this is not entirely true because React allows us to return an Array of elements as long as each element has a unique key.

How can we return adjacent elements:

```
class XYZ extends React.Component {
    render ( ) {
        return ( [ <div key='a' > Hello </div>,
                   <div key='b' > World </div>,
                   <div key='c' > Hi </div>
        ] )
    }
}
```

There is another feature build into React for this. That doesnot generate any HTML in DOM. It works as a higher order component.

- Create a new folder in src called hoc

  hoc stands for higher order components

- inside hoc folder create a file named: Aux.js

  Aux.js will contain following code.

```
const Aux = props => props.children

export default Aux
```

- usage : we use Aux component as a wrapper.

- using React.Fragment:

React 16 comes with with Aux component build in called: React.Fragment. Fragment component has same use case as our self created Aux component.

• Higher order Components (Introduction)

Components which wraps another components & may add some extra features to existing wrapped components.

features can be extra JSX
- ↳ styling
- ↳ extra logic

CONVENTION:
naming of hocs starts with With.....name.....

• There are two forms of defining Higher order components

1) Normal React Components
2) Normal Javascript based functions

• Normal React Component based HOC: suppose we create a HOC that adds styling (add more JSX)

HOC > WithClass.js

```
const WithClass = (props) =>
  ( <div className={props.classes}>
      {props.children}
    </div>
  )

export default WithClass
```

App.js

```
import WithClass from './hoc/...'

const App = ( ) => {
  return (
    <WithClass classes={styles}>
      <div>
        <h1> Hello World </h1>
      </div>
    </WithClass>
  )
}
```

- Higher Order Components using Normal JS Functions: use when to add logic

hoc > withClass.js

```
const withClass ( WrappedComponent , classes) {
    return (Props) => {
        <div className= { classes} >
            < WrappedComponent {...props} />
        </div>
    }
)
};
export default withClass;
```

App.js

```
import withClass from './hoc/..';

Const App = ( ) => {
    return (
        <div>
            <h1> Hello World </h1>
        </div>
    )
}

export default withClass(App, center)
```

- Setting State that depends upon previous state:

In many situations our new state depends upon previous state eg.

let us consider counter example as follows

src > containers > App2.js > default

```
1  import React from "react";
2
3  class App2 extends React.Component {
4    state = {
5      count: 0,
6    };
7
8    handleIncrement = () => {
9      this.setState({ count: this.state.count + 1 });
10   };
11
12   render() {
13     return (
14       <div>
15         <input value={this.state.count} />{" "}
16         <button onClick={this.handleIncrement}> Increment</button>
17       </div>
18     );
19   }
20 }
21
22 export default App2;
23
```

Problem with this Approach:

The problem with this approach is that although setState seems synchronous but actually it is not. React decides when to update state internally based on resource availability.

Thus even though setState might seem synchronous in reality it is not guaranteed that this.state.— holds true. consistent value.

Hence for situations where new state relies on previous state. we use another approach. setState takes a function as follows

```
this.setState((prevState, props) => {
    return {
        count: prevState.count + 1
    }
})
```

• PropTypes in React

Need for PropTypes:

Suppose we are working on a library or in a collaborative environment then people can use our components incorrectly. Thus it would be beneficial if we can provide some sort of instructions about what props a component accepts and type of props.

This can be achieved using extra package called PropTypes.

- Installation : npm install --save prop-types
- Import : import PropTypes from 'prop-types'
- Usage : after component definition

Counter.propTypes = { → is a special property for any JS object that React watches out for.

Counter → component Name

handleIncrement: PropTypes.func,

count: PropTypes.number → prop name

}

- Accessing DOM via React : refs

We can select any element in DOM using refs or references

On any element we can add special property ref.

These are couple of ways of using refs

- Function Based : oldest approach.

```
class App extends Component {
  render() {
    return (
      <div>
        <input value = "Hello" disabled = true.
          ref = {(inputEl) => this.inputElement = inputEl} />
      </div>
    )
  }

  componentDidMount() {
    this.inputElement.focus()
  }
```

ref takes in a function using which we set a global component level property inputElement

we can use this element inside component. didMount lifecycle Method.

However this works only in class based components and not on functional components

- **React.createRef():** Previous functional approach that we discussed was used in older versions of React. React provides us with createRef() which is used as follows. This is more Modern approach

```
class App extends React.Component {

    inputElementRef = React.createRef();

    render() {
        return (
            <div>
                <input ref={this.inputElementRef} />
            </div>
        )
    }

    componentDidMount() {
        console.log(this.inputElementRef.current)
    }
}
```

- **Refs in functional Components with React Hooks:**

Incase of functional components we cannot use React.createRef() Rather we have to make use of a React Hook known as useRef()

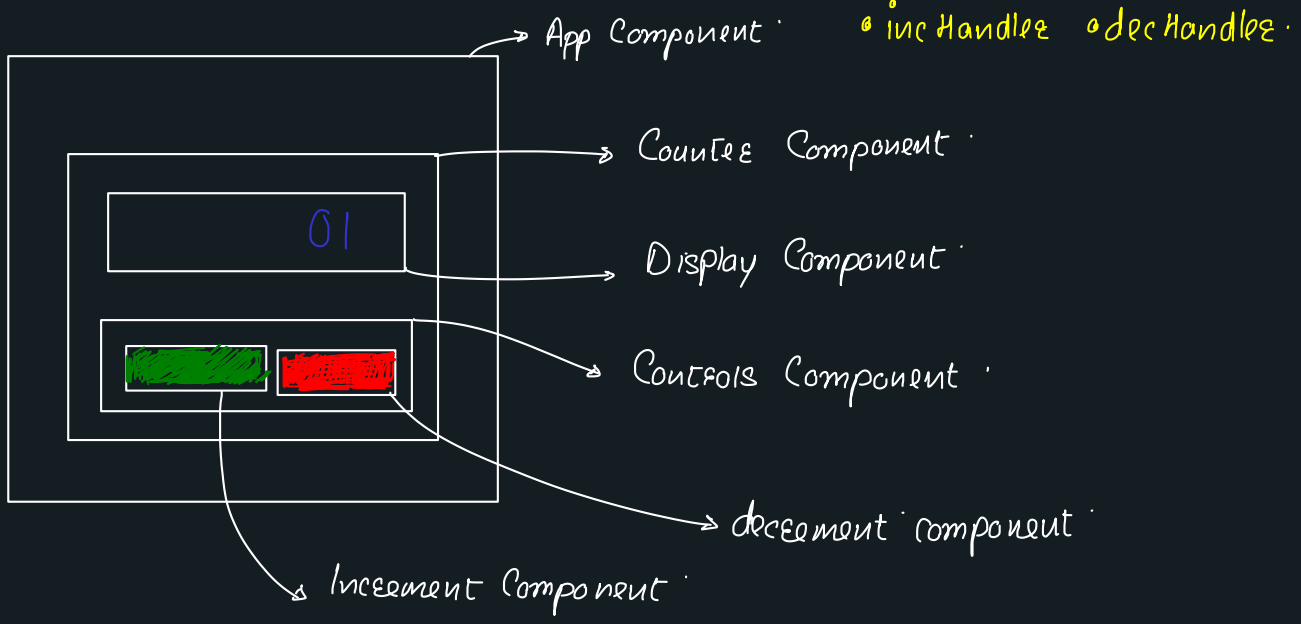
Following example will help us to understand

```
import React, { useRef } from 'react';

const App = (props) => {
    const mainRef = useRef (null);
    return (
        <div>
            <div ref = {mainRef}> Hello World </div>
        </div>
    )
}
```

Understanding Prop Chain Problem:

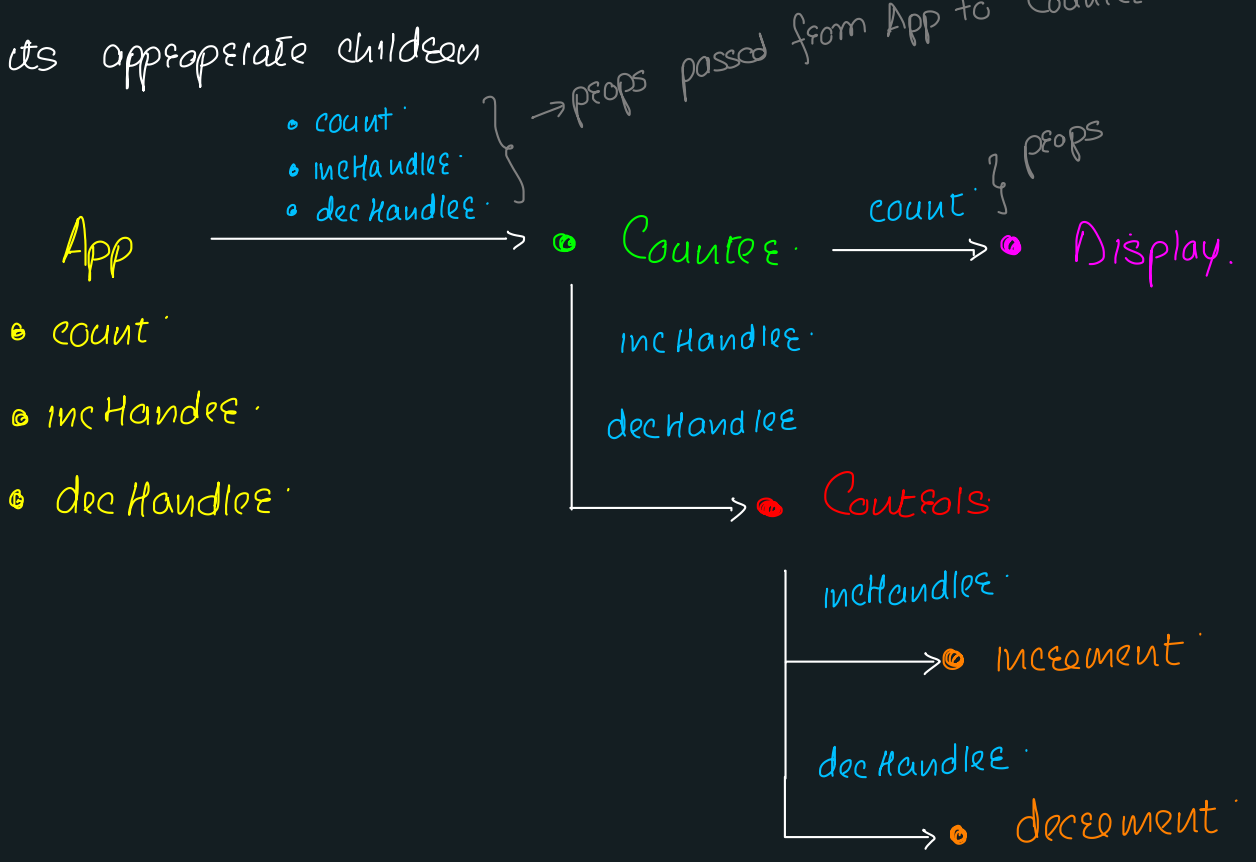
let us try to consider prop chain problem using a simple example. Suppose we have a counter App which can be disected as follows:

state = {count: 0}



App component maintains state & contains handlers

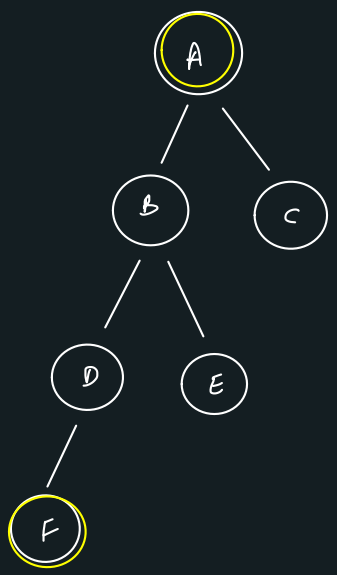
For this Application to work properly App needs to pass count and Handlers as props to its child component and child component to its appropriate children



Thus for increment component to use incHandler it needs to trace back a chain of props from Controls → Counter → App

Although for shallow components this might not be an issue but for deeply nested components it can be tedious.

To help with problem chaining React provides us with CONTEXT API. Context API provides us to pass props from a root component to deeply nested child component without need to pass props through the entire hierarchy eg.



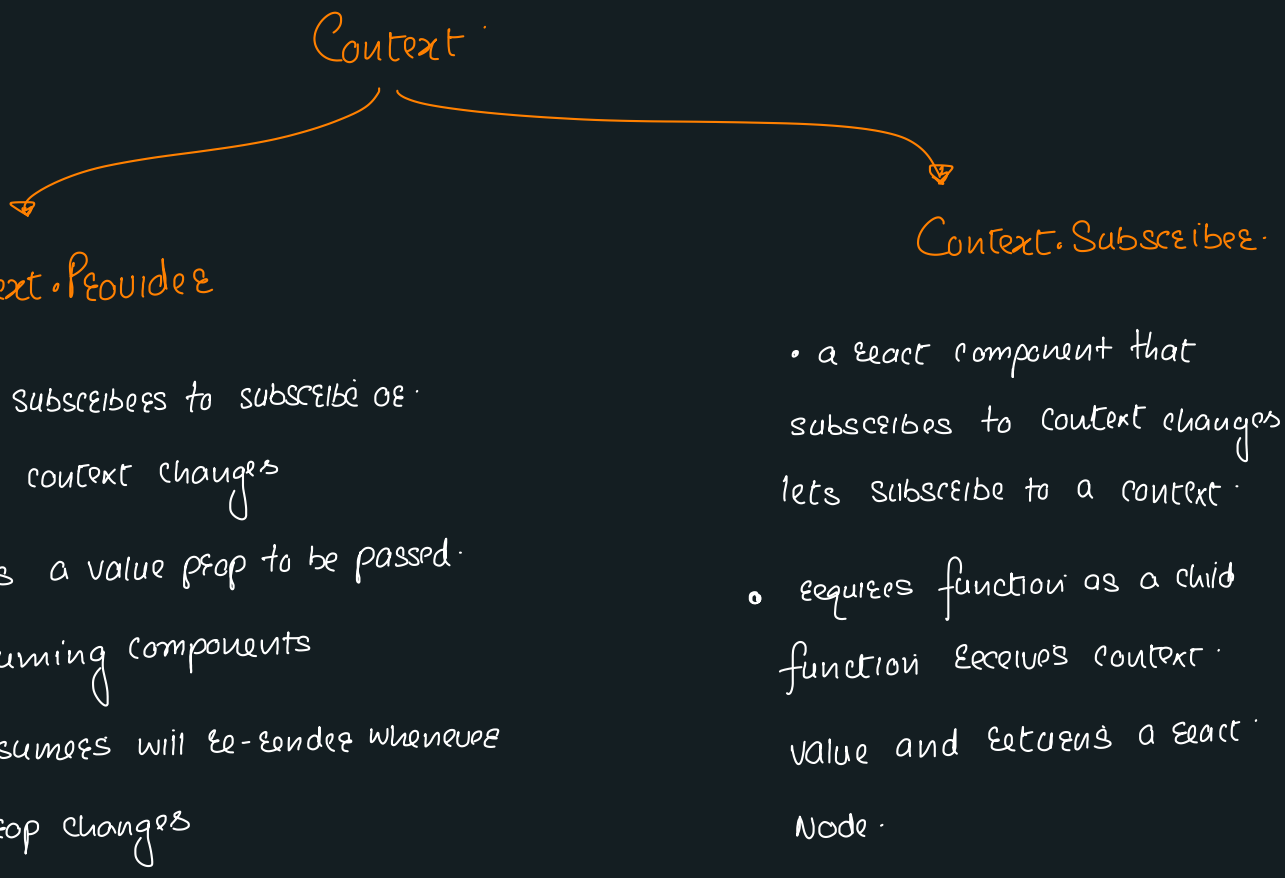
context API allows to pass props directly from A to F without need of passing props through B and D in this case.

Context (Documentation from Reactjs.org)

Context provides a way to pass data through component tree without having to pass props manually at every level.

API for Context - React.createContext

Context is based on a Provider Subscriber Model. Thus it has 2 parts
- Provider
- Consumer.



1. Creating context in src > context

src > context > CounterContext.js > ...

```
import React from "react";

const CounterContext = React.createContext({
  count: 0,
  incHandler: () => {},
  decHandler: () => {},
  clearHandler: () => {},
});

export default CounterContext;

```

2. Establishing Context . Provider

src > components > Counter > Counter.js > Counter > render > clearHandler

```
import React, { Component } from "react";

import Display from "./Display/Display";
import Controls from "./Controls/Controls";

import CounterContext from "../../context/CounterContext";

export default class Counter extends Component {
  state = {
    count: 0,
  };

  incHandler = () => {
    this.setState((prevState, props) => {
      return {
        count: prevState.count + 1,
      };
    });
  };

  decHandler = () => {
    this.setState((prevState, props) => {
      return {
        count: prevState.count - 1,
      };
    });
  };

  clearHandler = () => {
    this.setState((prevState, props) => {
      return {
        count: 0,
      };
    });
  };

  render() {
    return (
      <div style={styles}>
        <CounterContext.Provider
          value={{
            count: this.state.count,
            incHandler: this.incHandler,
            decHandler: this.decHandler,
            clearHandler: this.clearHandler,
          }}>
          <Display />
          <Controls />
        </CounterContext.Provider>
      </div>
    );
  }
}
```

Provider → value

Only these components can access context

src > components > Counter > Display > Display.js > Display > render

```
import React, { Component } from "react";

import CounterContext from "../../../context/CounterContext";

export default class Display extends Component {
  render() {
    return (
      <div style={styles}>
        <CounterContext.Consumer>
          {(context) => {
            return (
              <input
                style={styles.input}
                value={context.count}
                onChange={() => {}}
              />
            );
          }}
        </CounterContext.Consumer>
      </div>
    );
```

Consumer . function

src > containers > App.js > App > render

```
import React from "react";

import Counter from "../components/Counter/Counter";

class App extends React.Component {
  render() {
    return (
      <div
        style={{
          height: "100vh",
          width: "100vw",
          display: "flex",
          justifyContent: "center",
          alignItems: "center",
        }}>
        <Counter />
      </div>
    );
  }
}

export default App;

```

src > components > Counter > Controls > Controls.js > Controls > render

```
import React, { Component } from "react";

import CounterContext from "../../../context/CounterContext";

export default class Controls extends Component {
  render() {
    return (
      <div style={styles.wrapper}>
        <CounterContext.Consumer>
          {(context) => {
            return (
              <div>
                <button onClick={context.incHandler} style={styles.btnGreen}>
                  INC
                </button>
                <button onClick={context.decHandler} style={styles.btnOrange}>
                  DEC
                </button>
                <button onClick={context.clearHandler} style={styles.btnRed}>
                  CLEAR
                </button>
              </div>
            );
          }}
        </CounterContext.Consumer>
      </div>
    );
  }
}

const styles = {
  wrapper: {
    display: "flex",
    marginTop: "30px",
    justifyContent: "center",
  },

  btnRed: {
    padding: "15px 20px",
    backgroundColor: "red",
    margin: "auto 10px",
  },
  btnGreen: {
    padding: "15px 20px",
    backgroundColor: "green",
    margin: "auto 10px",
  },

  btnOrange: {
    padding: "15px 20px",
    backgroundColor: "orange",
    margin: "auto 10px",
  },
```

• ContextType :

- works with class based components only.
- provides more elegant way of writing consumer.

Previously

```
<Context.Consumer>
  {(context) => {
      ...
  }}
</Context.Consumer>
```

context is only accessible inside this block

- Hence we cannot use context in lifecycle methods or outside this block.

with ContextType :

src > components > Counter > Display > Display.js > Display > render

```
1  import React, { Component } from "react";
2
3  import CounterContext from "../../../context/CounterContext";
4
5  export default class Display extends Component {
6    static contextType = CounterContext;
7
8    componentDidMount() {
9      console.log(this.context);
10   }
11
12   render() {
13     return (
14       <div style={styles}>
15         <input
16           style={styles.input}
17           value={this.context.count}
18           onChange={() => {}}
19         />
20       </div>
21     );
22   }
23 }
24
```

① assign, here static is used so React can directly use contextType properly with creating an object of class.

② usuage all value of provider can be used as this.context.

Thus contextType increases scope of context inside consumer component.

Context for Functional Components :

incase of functional components, context is provided by useContext Hook

```
import React from 'react';
import CounterContext from '../../context/CounterContext';   (1) import Context.
import { useContext } from 'react';   (2) import Hook.

const Display = (props) => {
    const counterContext = useContext(CounterContext);   (3) bind context.
    return ( <div>
        <input value = { counterContext.count }   (4) use context.
          disabled = true />
        </div>
    )
}
```