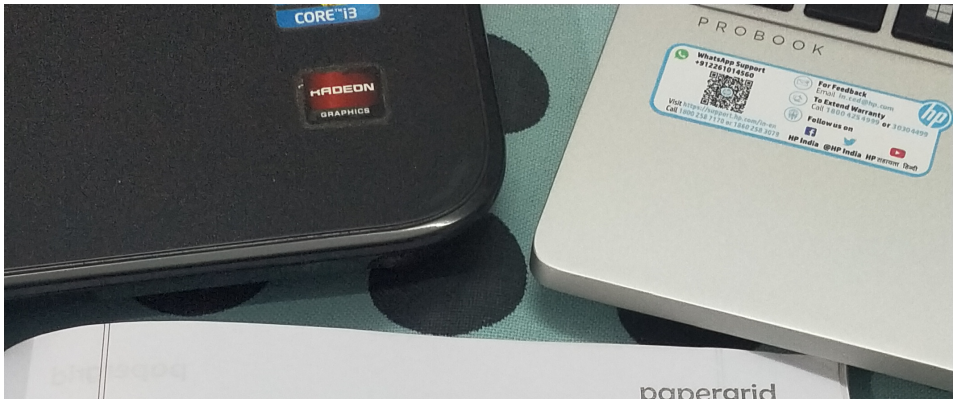


papergrid

Date: / /

React JS

-by Mosh.

- npm i -g create-react-app@1.5.2x
- src and public folder did not get created, so got this solution.
  - 1) npm rm -g create-react-app
  - 2) npm install -g create-react-app
  - 3) npx create-react-app my-app

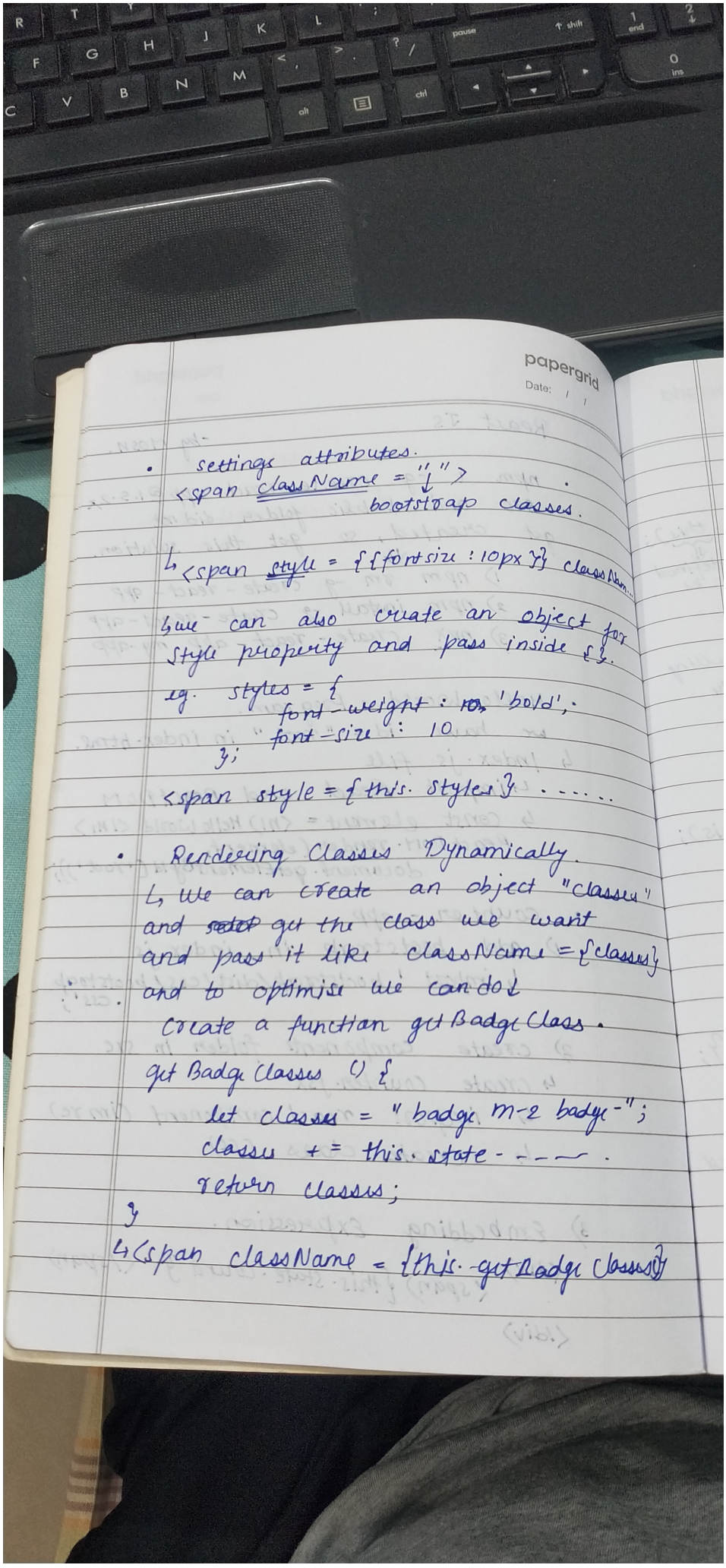
- Hello World - Program.
  We have :id = "root" in index.html.
  - Index.js file
    - import React and ReactDOM
    - const element = <h1>Hello World</h1>
      ReactDOM.render(element, document.getElementById('root'));

- counter-app
  1) add bootstrap in index.js
     - import 'bootstrap/dist/css/bootstrap.css';

  2) create components folder in src
     - create counter.jsx
       - import react component (imrc)
       - create class (cc)

3) Embedding Expression.

```
<div>
    <span> {this.state.count} </span>
</div>
```



papergrid
Date: / /

- settings attributes.

bootstrap classes.

↳ <span style = {{fontsize : 10px}} classNam...

↳ we can also create an object for style property and pass inside {}.

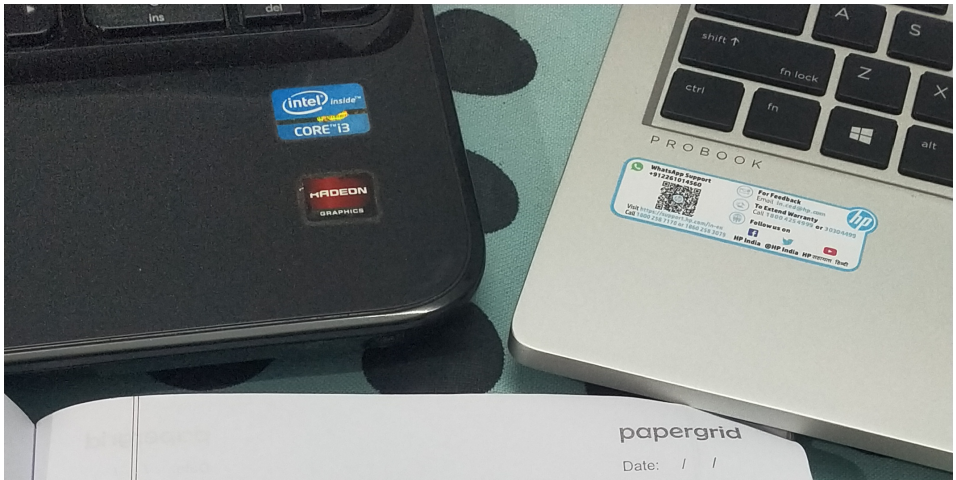
eg. styles = {
font-weight : ~~10~~ 'bold',
font-size : 10
};

<span style = {this.styles} .....

- Rendering Classes Dynamically.
↳ we can create an object "classes" and ~~select~~ get the class we want and pass it like className = {classes}
and to optimise we can do↓
Create a function getBadgeClass.

```
getBadgeClasses() {
    let classes = "badge m-2 badge-";
    classes += this.state - - - -.
    return classes;
}
```

↳ <span className = {this.getBadgeClasses()}



papergrid

Date: / /

- Rendering a list in React
  - in Angular, we have ngFor.
- state = {
  tags : ["tag1", "tag 2", "tag 3"]
  };

{this.state.tags.map(tag => <li key={tag}>{tag})}

(arrow function)

- Conditional Rendering.
  - Create a method renderTags() {
    if (this.state.tags.length === 0) return
    There are No Tags
    return {this.state.tags.map .....
    render () {
    return {this.renderTags()}
    }

OR

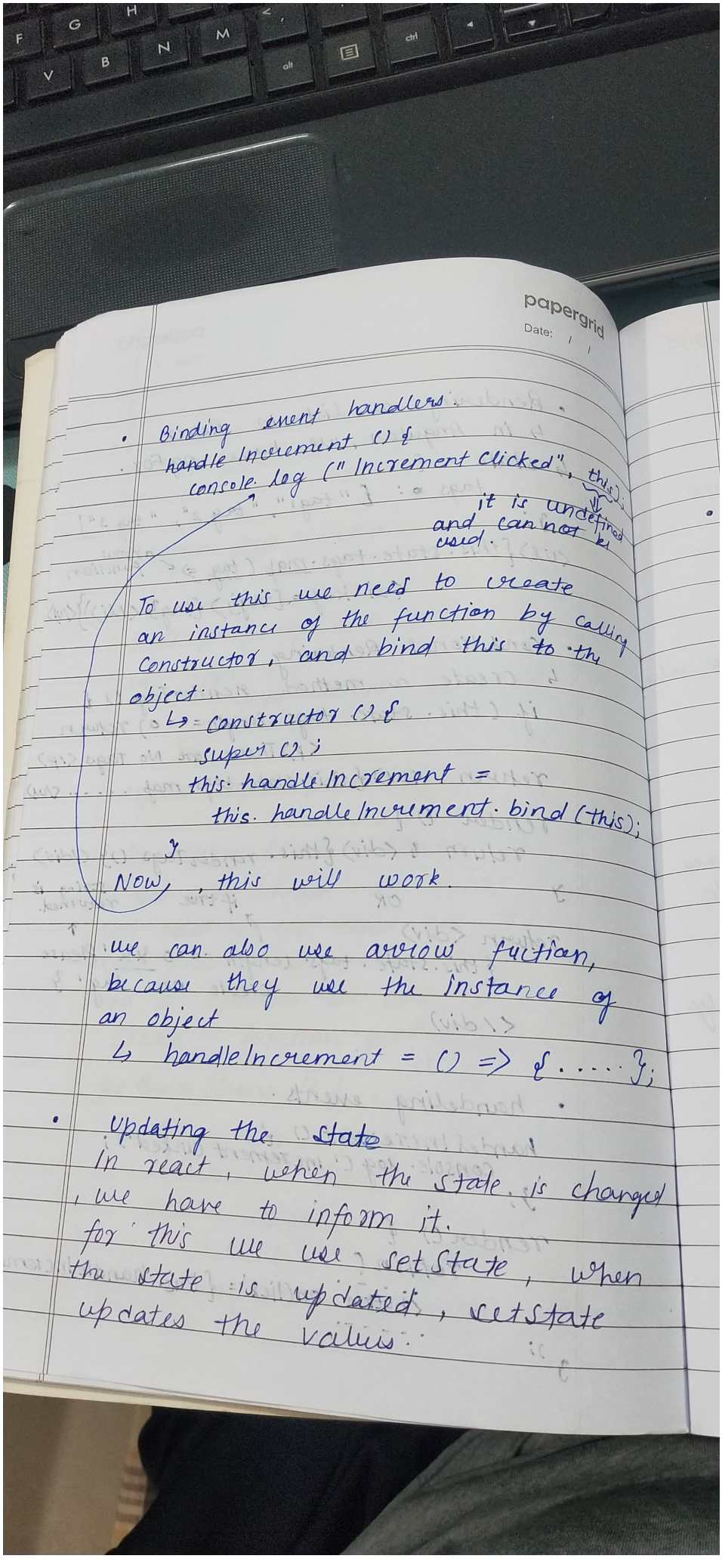
return
{this.state.tags.length === 0 && 'Please create new tag'}

(if true → string is returned.)

- handeling events.

handelIncrement() {
console.log('Increment Clicked');
};

render() {
return (
<button onClick={this.handleIncrement}
.....
);
}



papergrid
Date: / /

- Binding event handlers.

```
handleIncrement() {
  console.log("Increment clicked", this);
}
```

this → it is undefined and can not be used.

To use this we need to create an instance of the function by calling constructor, and bind this to the object.

↳
```
constructor() {
  super();
  this.handleIncrement =
    this.handleIncrement.bind(this);
}
```

Now, this will work.

we can also use arrow fuction, because they use the instance of an object

↳ `handleIncrement = () => {....};`

- Updating the state

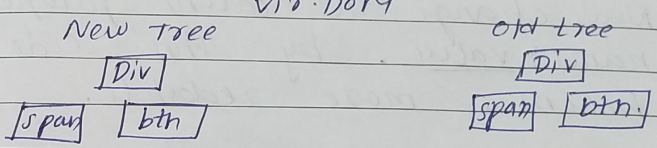
In react, when the state is changed, we have to inform it.
for this we use setState, when the state is updated, setState updates the values.

papergrid

Date: / /

↳ handleIncrement = () ⇒ {
this.setState({count: this.state.count +1})
};

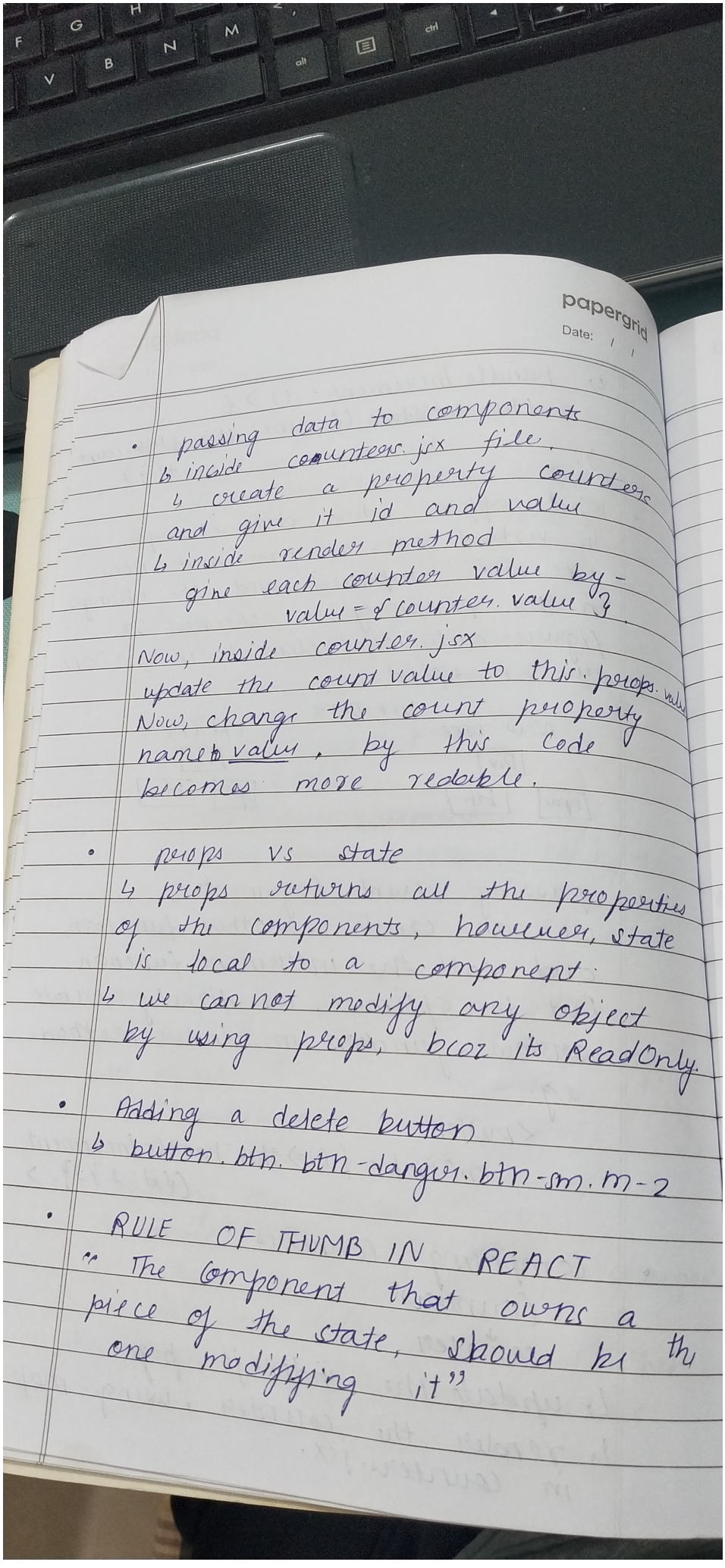
- What happens when state changes.
In virtual DOM, old tree and new tree are compared, and the changes in the property is checked to figure out what elements in vir.DOM are modified.



- Passing event Arguments.
we can create, another function, and call the increment function.
But to optimise, we direcly create arrow function in render method.
eg.

```
<button
  onClick={() ⇒ this.handleIncrement({id: 1})} >
```

- Composing Counters
Counters
counter
↳ update the index.js page.
↳ render the counter, using map in counters.jsx.



papergrid
Date: / /

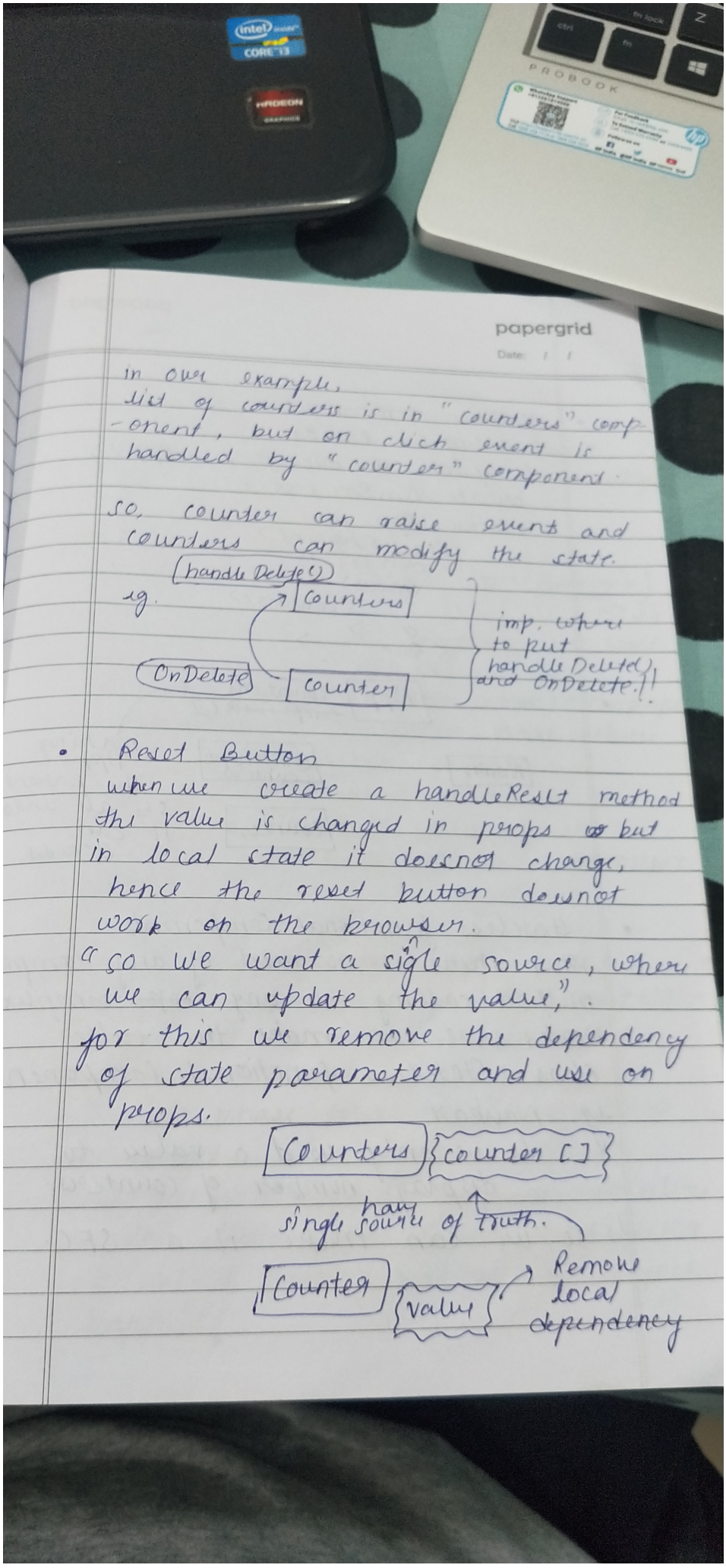
- passing data to components
  - inside counters.jsx file.
    - create a property counters and give it id and value
    - inside render method
      give each counter value by -
      value = {counter.value}.

Now, inside counter.jsx
update the count value to this.props.value
Now, change the count property name to value, by this code becomes more readable.

- props vs state
  - props returns all the properties of the components, however, state is local to a component.
  - we can not modify any object by using props, bcoz its ReadOnly.

- Adding a delete button
  - button.btn.btn-danger.btn-sm.m-2

- RULE OF THUMB IN REACT
  " The Component that owns a piece of the state, should be the one modifying it"



papergrid

Date: / /

in our example,
list of counters is in "counters" component, but on click event is handled by "counter" component.

so, counter can raise events and counters can modify the state.

- Reset Button

when we create a handleReset method the value is changed in props but in local state it doesn't change, hence the reset button doesn't work on the browser.

"so we want a single source, where we can update the value".

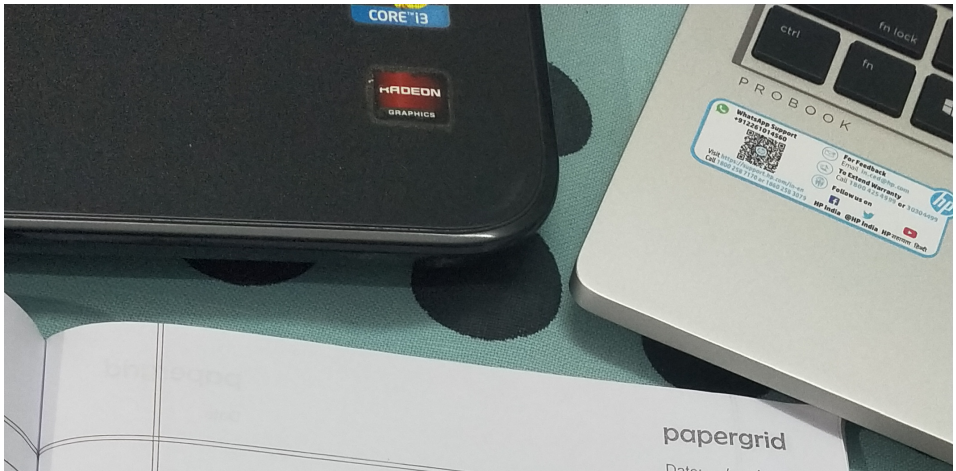
for this we remove the dependency of state parameter and use on props.

papergrid
Date: / /

- Multiple components.
  ↳ adding a NavBar
  ↳ update ~~the~~ index.js, <App />
  create NavBar.jsx.
  Inside App.js
  <React.Fragment>
  <NavBar />
  <main>
  <Counters />
  </main>
  </React.Fragment>

- App — counters ~~components~~ []
  NavBar
  ~~Components~~ Counters
  Counter
  making ~~App~~ the parent so, all under it can access.

- Stateless functional components.
  When there is no need of any complex method calling or any ~~copt~~ complex work, we can make the ~~catt~~ class → Stateless functional component.
  eg. Navbar
  ↳ we only used a value to display number of counters.
  ↳ so we can make it a SFC.



papergrid

Date: / /

- Destructuring Arguments.
  ↳ sometimes when working with complex apps, when we call something like props. counter. counters.... it becomes less optimized.
  So, we can do this -
  ① const NavBar = ({totalCounters}) => {}
  the property we want to call.
  ② const {onDelete, onIncrement..etc} = this.props and remove this.props from where it is used.

- Life Cycle Hooks.

MOUNT	UPDATE	UNMOUNT
constructor		
render	render	
componentDid-Mount	componentDid-Update	componentWill-Unmount.

- mount phase
  in this phase first constructor (LCH) is called.
  ↳ this is a great time to initialize the state coz when constructor is called initialization of object happened.

then render
↳ In this phase all the components are rendered in the browser.
then componentDidMount
↳ after all the components are rendered, its great time to ~~check if prevState or prevProps~~ call the AJAX call and take the data.

- Update phase.
↳ When any setState or event happens, render is called.
↳ After that componentDidUpdate is called, it great time to check if prevProp or prevState is changed, and we can make the AJAX call for data.

- Unmounting phase
In unmounting, when a component (needed to be removed) is removed from DOM, we have a new virtual DOM, react will compare it with old virtual DOM, It will call the componentWillUnmount. It helps from memory leaks.