# React Notes

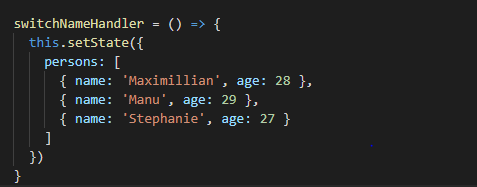
* Render: React uses this method to render jsx code to screen
* React.createElement(‘div’, {className: ‘class-a’}, ‘text’); React.render uses React.createElement in the background
* State is available only in class based components, not in function based components.

State is used to set component internal properties which can be used with *this.*

* Props are used to get input properties of component
* Actually, only changes in props  and/ or state  trigger React to re-render your components and potentially update the DOM in the browser

## setState

setState method is used to change state of the component in class based components.



In case the state is to be updated on the basis of old state in that case provide a function to setState method with prevState and prevProps as arguments

this.setState((prevState, prevProps) => {

return {

persons: persons,

changeCounter: prevState.changeCounter + 1

}

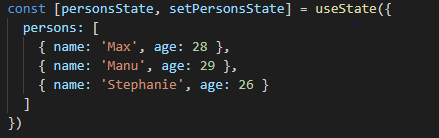
})

## useState

**useState** method is a react hook which returns array of two elements:

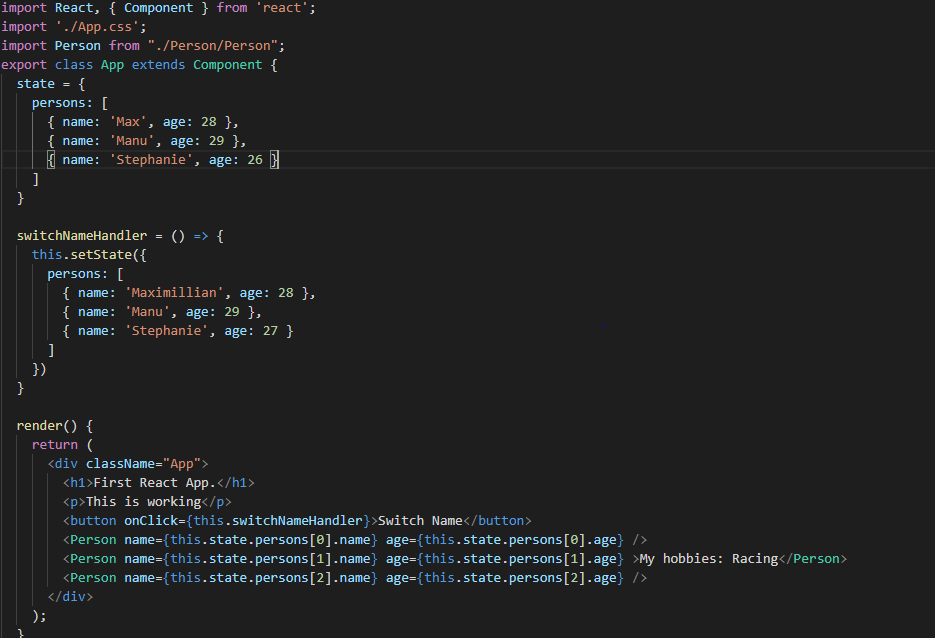
1: current state

2: method to update the state



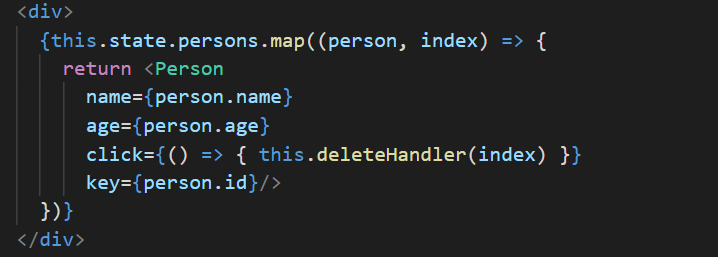
Updating the state with useState case will completely replace the old state

## Class Based Component



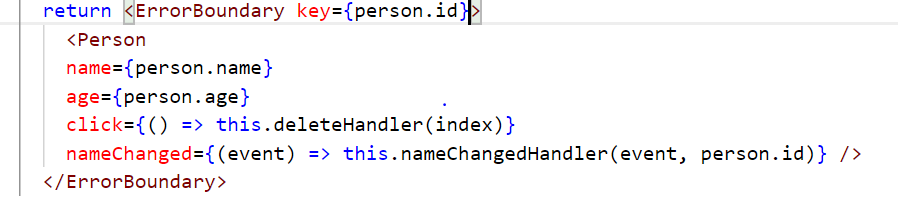
## Function based component

## Printing lists

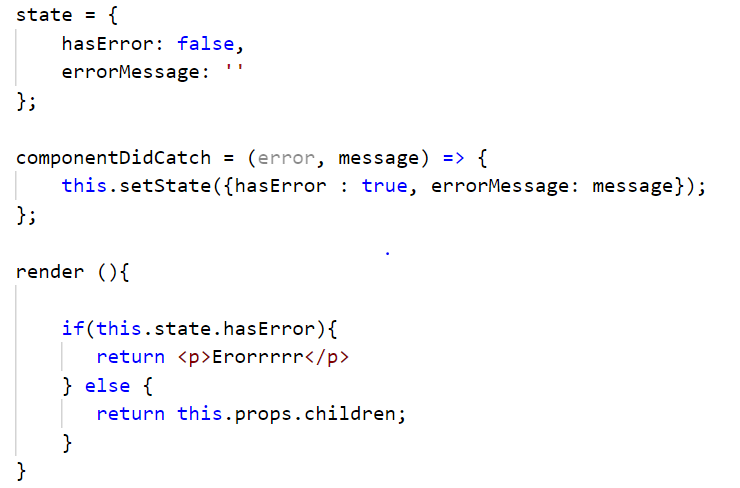


## Error Boundaries

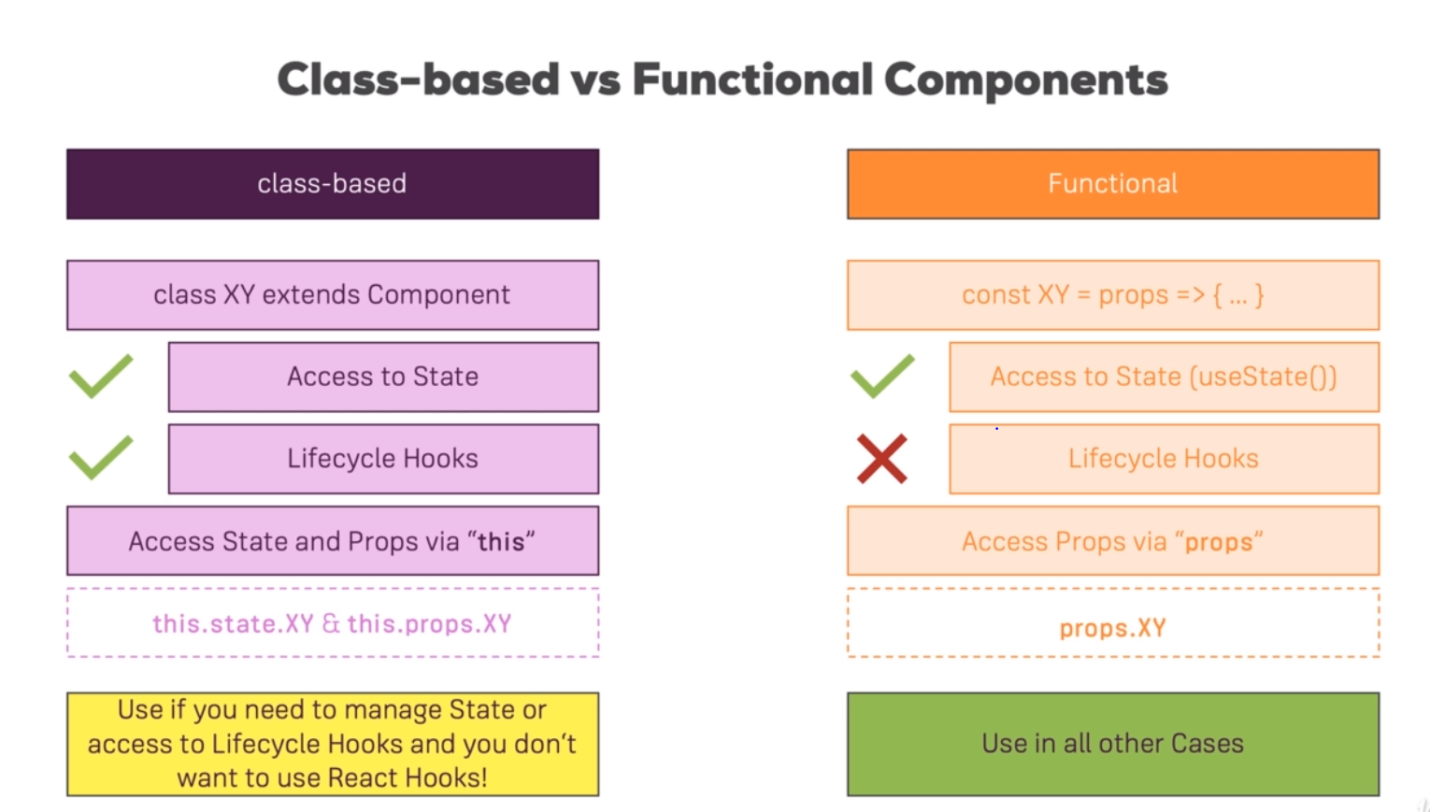
Creating error handler for components. Component which might throw error should be wrapped inside ErrorBoundary Component like shown below.



ErrorBoundary Component.js

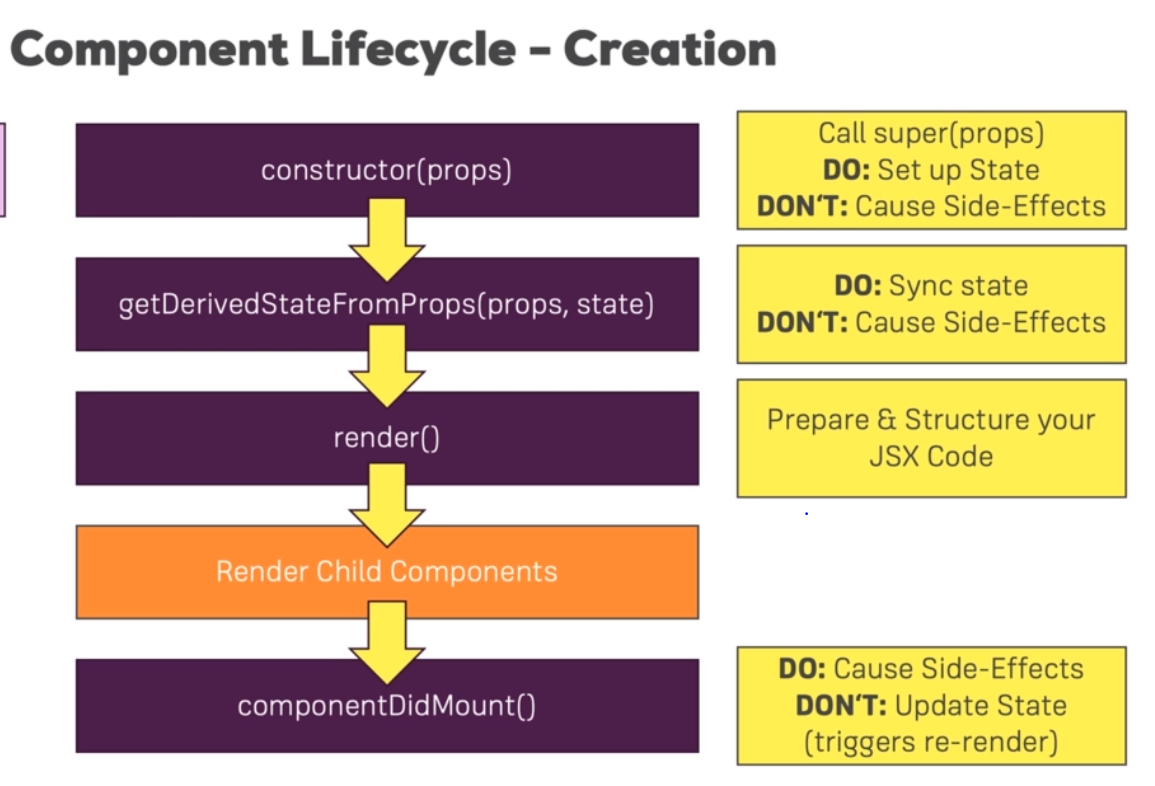


## Class based VS function based components

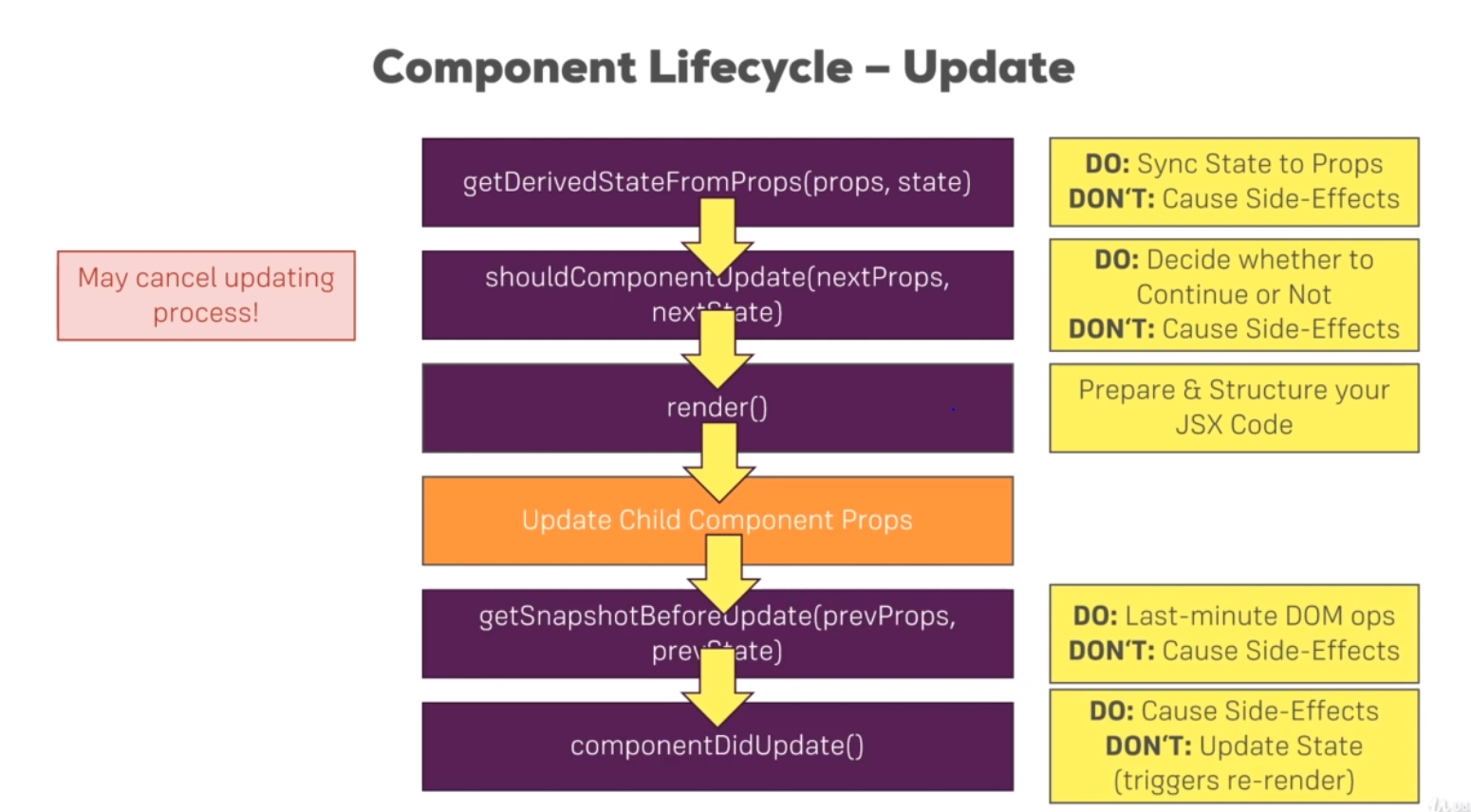


## Component Lifecycle – Creation

1. Constructor:
2. getDerivedStateFromProps(props, state): Whenever props change we can use this lifecycle hook to sync state with props and return the updated state (hook is used very less frequently)
3. render(): Used to prepare and return JSX code. Should not be used to make http calls or timeout functions. If child components are used inside render method then all the child components will be rendered and only when all the children have rendered, our current render method will be completed
4. componentDidMount(): After render has finished. Can be used to create side effects like making http request, timeout function, etc. Should not change state as it will re-trigger render method



## Component update lifecycle



## shouldComponentUpdate

Decide whether a class based component should update or not

shouldComponentUpdate(nextProps, nextState) {

console.log('[Person.js] shouldComponentUpdate');

return nextProps.persons !== this.props.persons

}

Implement pureComponent instead of shouldComponentUpdate lifecycle hook in case you want to apply a check on all of the props

## useEffect

useEffect is a react hook which takes a function as an argument and executes it for every render cycle.

It is a combination of lifecycle hooks: componentDidMount and componentDidUpdate

Run after each render cycle

useEffect(() => {

console.log('[cockpit.js] useEffect');

});

Run only after first render cycle and when any of the dependencies change

Give second argument which specifies when to run useEffect. Here useEffect will only call the method when props.persons changes, otherwise it will run with each render cycle

useEffect(() => {

console.log('[cockpit.js] useEffect');

}, [props.persons]);

Only run on mount and unmount

useEffect(() => {

console.log('[cockpit.js] useEffect');

}, []); This runs when component is mounted for the first time and when component is unmounted

Component will unmount is implemented by returning a function. That returned function will be called only when component is about to be unmounted

useEffect(() => {

console.log('[cockpit.js] useEffect');

// If you want to implement componentWillUnmount,

// return a function from here, and React will call it after each render cycle and before actual useEffect function

return () => { console.log('cleanup work');}

});

## Memo

Wrap your component inside react.memo() so that your component doesn’t need to update with every change in any of the parent components unless and until current component’s properties change. It is used in function based components and is similar to shouldComponentUpdate() in class based components.

export default React.memo(Cockpit);

## propTypes

install package: npm install –save prop-types

To defined the type of props a component expects and should be given to it

Install: npm install –save prop-types

class Person extends Component {

render() {

console.log('[Person.js] rendering...')

return (

<Aux>

<p onClick={this.props.click}>I am {this.props.name} and I am {this.props.age} years old.</p>

<p>{this.props.children}</p>

<input type="text" onChange={this.props.nameChanged} value={this.props.name}></input>

</Aux>

)

};

}

Person.propTypes = {

click: PropTypes.func,

name: PropTypes.string,

age: PropTypes.number,

nameChanged: PropTypes.func

}

export default withClass(Person, classes.Person);

## Ref

Reference to an element on DOM

<input type="text" ref={this.inputElementRef} onChange={this.props.nameChanged} value={this.props.name}></input>

For class based components use createRef() method of react

this.inputElementRef = React.createRef(); // Create placeholder for reference

For function based components use ‘useRef’ react hook

const toggleBtnRef = useRef(null); // create ref for button element in a function based component

## React.createContext()

To avoid unnecessary chain of passing of props we can use createContext(). Context is a javascript object which is available where we want it to be available

import React from 'react';

const authContext = React.createContext({

authenticated: false,

login: () => { }

}); // Globally available javascript object, the values provided here are default ones and they come into play when values have not been set explicitly from AuthContext.Provider

export default authContext;

Provider: Context will be available only to the child elements of provider

<AuthContext.Provider value={{ authenticated: this.state.authenticated, login: this.loginHandler }}>

{this.state.showCockpit ? <Cockpit

title={this.props.appTitle}

personsLength={this.state.persons.length}

showPersons={this.state.showPersons}

toggle={this.togglePersonsHandler}>

</Cockpit> : null}

{person}

</AuthContext.Provider>

Consumer: Takes a function which has context as an argument and that context can be used

<AuthContext.Consumer>

{(context) => context.authenticated ? <p> Authenticated! </p> : <p> Please Login </p>}

</AuthContext.Consumer>

To use inside javascript code:

1. **Class Based components**: Create a static contextType property and assign the context object to it

static contextType = AuthContext;

now this can be used inside JSX like

{this.context.authenticated ? <p> Authenticated! </p> : <p> Please Login </p>}

1. **Function based components:** use useContext React hook

const authContext = useContext(AuthContext); // React hook to use context inside js code

## Routing

<Link to=”/” component={ComponentName}>

Using Link does not apply ‘active’ class to the currently active <a> tag. To have ‘active’ class applied to the currently active <a>, use <NavLink> instead.

“to” takes an absolute path by default

Route properties can be obtained by using props object, but are only available on the immediate component loaded on that path and are not passed down to the component tree.

Route properties can be obtained in the child components by two ways:

1. By passing as parameters to the selector by passing down props
2. <Post
3. key={post.id}
4. title={post.title}
5. author={post.author}
6. {...this.props}
7. clicked={() => this.postSelectedHandler(post.id)} />;
8. by using WithRouter HOC
9. import { withRouter } from 'react-router-dom';

export default withRouter(Posts);

Now router props will be available inside Posts component tree

## Saga

yield all([  
 WatcherSaga1(),  
 WatcherSaga2(),  
 ])

With all(), they will both start at the same time. But without “all” WatcherSaga2 will wait for WatcherSaga1 to finish

# SyntheticEvents

Synthetic events are a wrapper on the native DOM events so as to provide cross browser feasibility.

React uses a single event listener per single event type to invoke all event handlers which were attached in the virtual DOM. These event handlers are attached to the top of the actual DOM and make use of event delegation.

ForExample:

const ExampleComponent = () => (

<div onBlur=(onBlur)>

<div onClick={onClick}>

<div onClick={onClick} />

</div>

</div>

);

We will have a single event listener registered on the native DOM for the click event.

{click: Array(1), blur: Array(1)}

{

Since React is registering a single event listener per multiple handlers on the actual DOM, it would need to re-dispatch the event for each and every handler. React keeps a collection of actual event handlers and their corresponding fiber nodes(nodes in the virtual DOM tree). And calls the actual event handler from there.

# Enzyme methods

Mount: Full rendering including all hooks and children in render

Shallow: Rendering but no children

Render: Only render function with all children

## LazyLoading

const Artists = React.lazy(() => import('./Artists'));  
  
function MyComponent() {  
return (  
<div>  
<Suspense fallback={<div>Loading...</div>}>  
<Artists />  
</Suspense>  
</div>  
);  
}