Chapter-13

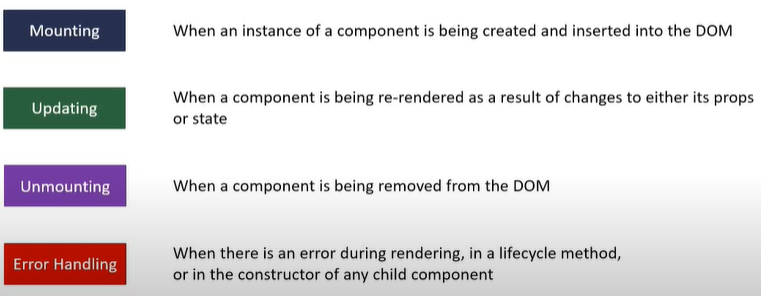
**Lifecycle methods in react**

* When we create react component, the component goes through several stages in its lifecycle.
* React provides with some built-in methods so that we can override at particular state in lifecycle.

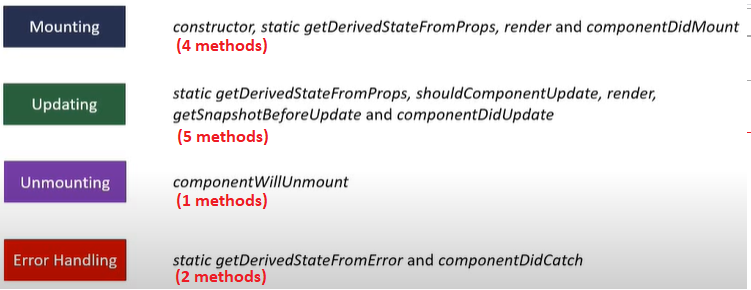
**Lifecycle methods available for class component** (these methods do not exist for functional component)

* Mounting
* Updating
* Unmounting
* Error handling

**When these methods are called?**



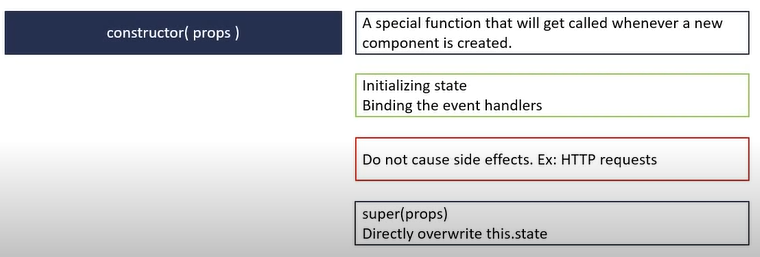
**Methods called during different phases**



**Mounting Lifecycle methods**

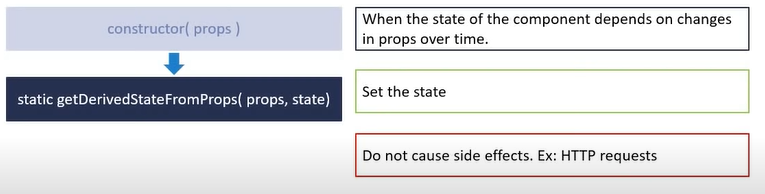
**constructor(props)🡪**

* It’s a special function that will get called whenever a new component is created.
* It is used for Initializing state and binding the even handler
* While defining our own constructor when need to call base class constructor using super(props) in our constructor.
* Constructor is the only place where we can directly overwrite the this.state otherwise at other places we have to use setState method.
* We should not cause side effect in constructor Ex. HTTP requests.



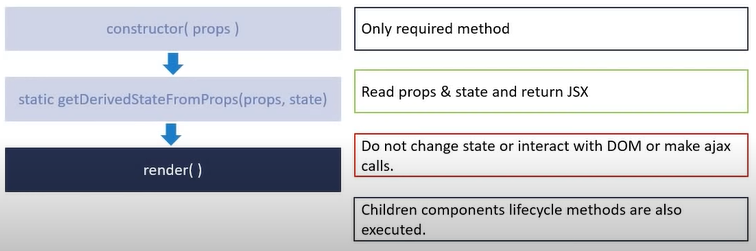
**static getDerivedStateFromProps(props, state)🡪**

* It’s a one of the rarely used methods (as per react docs)
* This method is used when state of the component depends on changes in props over time to set the state
* As this is the static method so it does not have access to “this” keyword, so here we simply return the object that represents the state of the component and do not use this.state.
* This method take props and state as parameter and has to return new state or null.
* We should not cause side effect in this method Ex. HTTP requests.



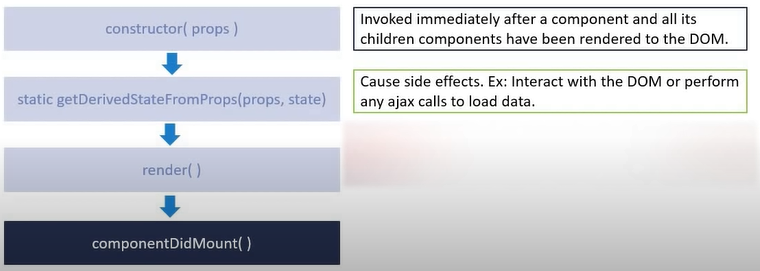
**render()🡪**

* It is the only required method in class a component.
* In this method we simply read the props and state and returns the jsx which represents the UI.
* We should not change the state or interact with DOM or make ajax calls.
* This may also contain other children components so right after the parent render method children lifecycle methods are also executed.



**componentDidMount()🡪**

* It is called only once in whole life cycle of given component.
* It invokes immediately after a component and all its children component have been rendered to the DOM.
* Its perfect place to cause side effect ex. Interact with DOM or perform any ajax calls to load data



import React, { Component } from "react";

class LifecycleA extends Component {

  constructor(props) {

    super(props);

    this.state = {

      name: "Abhi"

    };

    console.log(`LifecycleA constructor`);

  }

  //we have to add  static keyword otherwise method will be ignored

  //this take props and state as parameter and has to return new state or null

  static getDerivedStateFromProps(props, state) {

    console.log(`LifecycleA getDerivedStateFromProps`);

    return null;

  }

  componentDidMount() {

    console.log(`LifecycleA componentDidMount`);

  }

  render() {

    console.log(`LifecycleA render`);

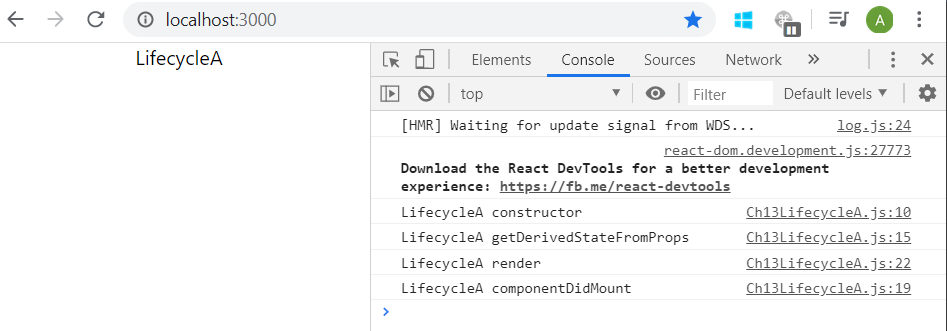
    return <div>LifecycleA</div>;

  }

}

export default LifecycleA;

OUTPUT🡪



When the component has children component

import React, { Component } from "react";

class LifecycleB extends Component {

  constructor(props) {

    super(props);

    this.state = {

      name: "Abhi"

    };

    console.log(`LifecycleB constructor`);

  }

  //we have to add  static keyword otherwise method will be ignored

  //this take props and state as parameter and has to return new state or null

  static getDerivedStateFromProps(props, state) {

    console.log(`LifecycleB getDerivedStateFromProps`);

    return null;

  }

  componentDidMount() {

    console.log(`LifecycleB componentDidMount`);

  }

  render() {

    console.log(`LifecycleB render`);

    return <div>LifecycleB</div>;

  }

}

export default LifecycleB;

import React, { Component } from "react";

import LifecycleB from "./Ch13LifecycleB";

class LifecycleA extends Component {

  constructor(props) {

    super(props);

    this.state = {

      name: "Abhi"

    };

    console.log(`LifecycleA constructor`);

  }

  //we have to add  static keyword otherwise method will be ignored

  //this take props and state as parameter and has to return new state or null

  static getDerivedStateFromProps(props, state) {

    console.log(`LifecycleA getDerivedStateFromProps`);

    return null;

  }

  componentDidMount() {

    console.log(`LifecycleA componentDidMount`);

  }

  render() {

    console.log(`LifecycleA render`);

    return (

      <div>

        <div>LifecycleA</div>;

        <LifecycleB />

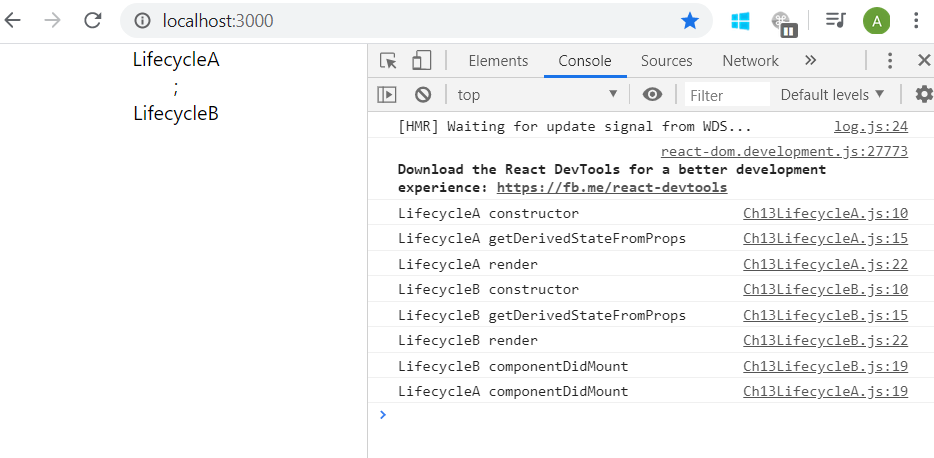
      </div>

    );

  }

}

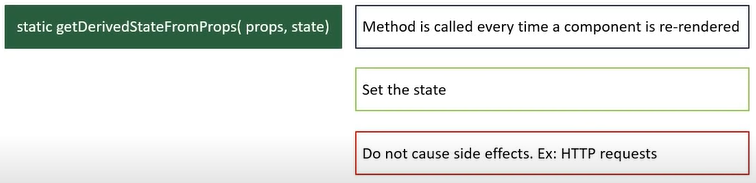
export default LifecycleA;



**Updating Lifecycle method🡪**

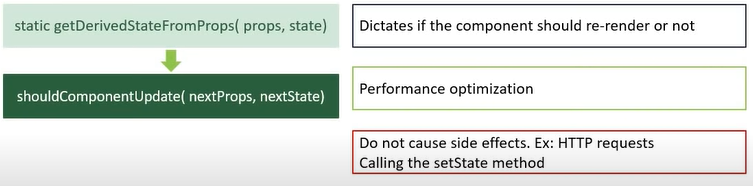
**1🡪 static getDerivedStateFromProps(props, state):**

As mounting lifecycle method:



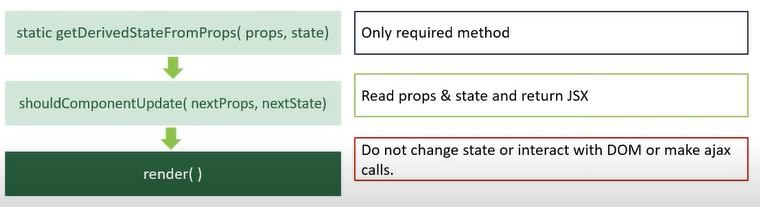
**2🡪 shouldComponentUpdate(nextProps, nextState):**

* It’s one of the rarely used lifecycle methods (as per react docs).
* It dictates if the component should re-render or not.
* By default all class component re-render when they receive a prop or their state changes.
* Here we can compare existing props and stats values with nextProps and nextState values and return true or false. If we are returning false and component won’t be re-render.
* It’s used for performance optimization.
* Here we should not cause side effects ex. HTTP requests.
* Here we should not call setState method.



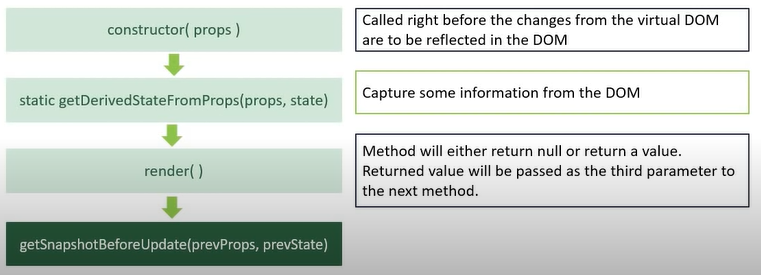
**3🡪 render():**

Also present in mounting lifecycle method



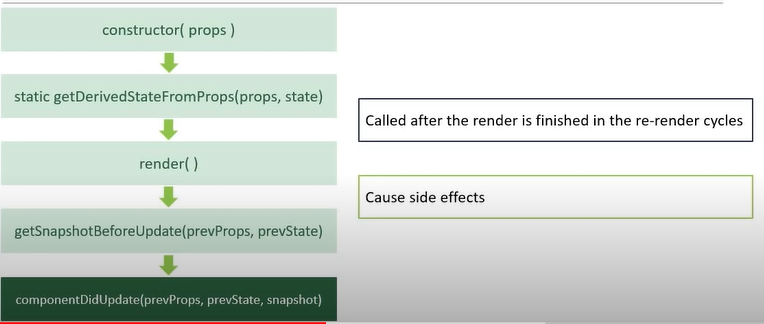
**4🡪getSnapshotBeforeUpdate(prevProps, prevState):**

* This is one the rarely used methods as per the react docs
* This method is called right before the changes from the virtual DOM are to be reflected in the DOM.
* It’s used to capture some information from the DOM ex. We can read the users scroll position and maintain the scroll positon after the update performing some calculation.
* This method will either return null or a value. Returned value will be passed as the third parameter to the next method.



**5🡪componentDidUpdate(prevProps, prevState, snapshot):**

* This method is called on rendering is finished in re-render cycle.
* This is guaranteed to be called only once in re-render cycle.
* We can cause side effects here ex. We can make ajax calls, for that 1st we should compare prevProps with new props and then decide whether to make ajax call or not.



import React, { Component } from "react";

class UpdatingLifeCycleB extends Component {

  constructor(props) {

    super(props);

    this.state = {

      name: "Abhi"

    };

    console.log(`UpdatingLifeCycleB constructor`);

  }

  componentDidMount() {

    console.log(`UpdatingLifeCycleB componentDidMount`);

  }

  static getDerivedStateFromProps(props, state) {

    console.log(`UpdatingLifeCycleB getDerivedStateFromProps`);

    return null;

  }

  shouldComponentUpdate() {

    console.log(`UpdatingLifeCycleB shouldComponentUpdate`);

    return true;

  }

  getSnapshotBeforeUpdate() {

    console.log(`UpdatingLifeCycleB getSnapshotBeforeUpdate`);

    return null;

  }

  componentDidUpdate() {

    console.log(`UpdatingLifeCycleB componentDidUpdate`);

  }

  render() {

    console.log(`UpdatingLifeCycleB render`);

    return <div>UpdatingLifeCycleB</div>;

  }

}

export default UpdatingLifeCycleB;

import React, { Component } from "react";

import UpdatingLifeCycleB from "./Ch13UpdatingLifecycleB";

class UpdatingLifeCycleA extends Component {

  constructor(props) {

    super(props);

    this.state = {

      name: "Abhi"

    };

    console.log(`UpdatingLifeCycleA constructor`);

  }

  static getDerivedStateFromProps(props, state) {

    console.log(`UpdatingLifeCycleA getDerivedStateFromProps`);

    return null;

  }

  componentDidMount() {

    console.log(`UpdatingLifeCycleA componentDidMount`);

  }

  shouldComponentUpdate() {

    console.log(`UpdatingLifeCycleA shouldComponentUpdate`);

    return true;

  }

  getSnapshotBeforeUpdate() {

    console.log(`UpdatingLifeCycleA getSnapshotBeforeUpdate`);

    return null;

  }

  componentDidUpdate() {

    console.log(`UpdatingLifeCycleA componentDidUpdate`);

  }

  changeState = () => {

    this.setState({

      name: "Minnu"

    });

  };

  render() {

    console.log(`UpdatingLifeCycleA render`);

    return (

      <div>

        <div>UpdatingLifeCycleA</div>

        <button onClick={this.changeState}>Change State</button>

        <UpdatingLifeCycleB />

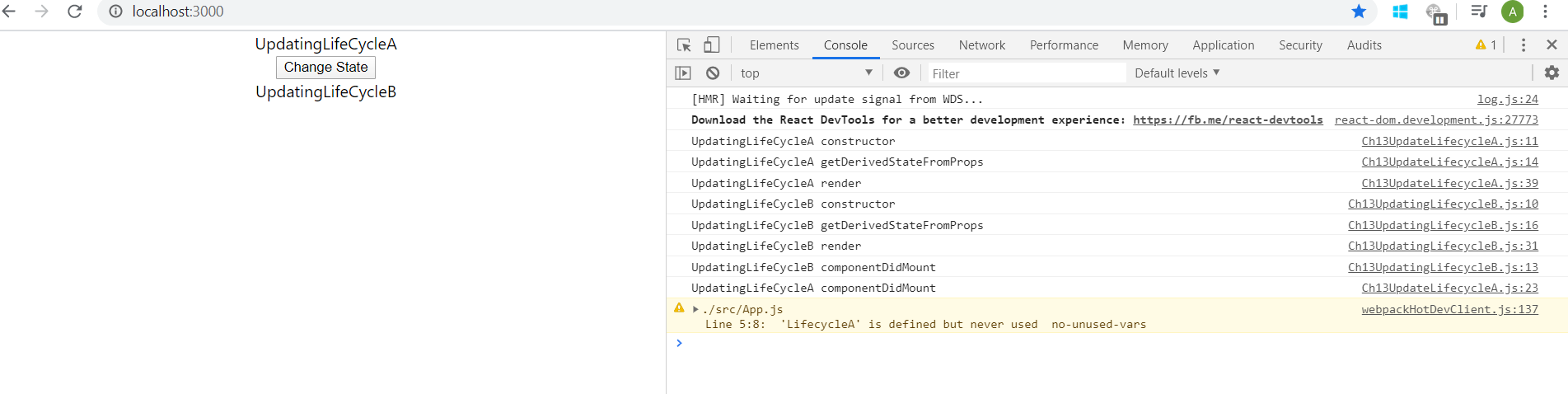
      </div>

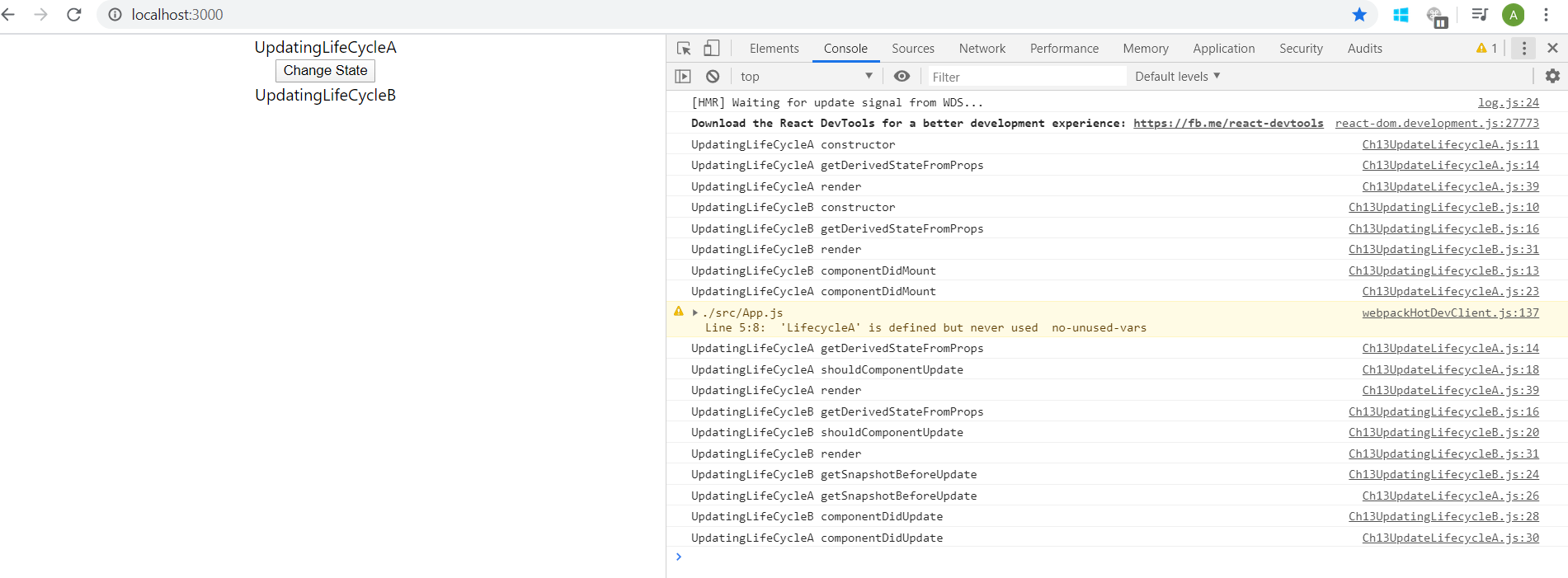
    );

  }

}

export default UpdatingLifeCycleA;

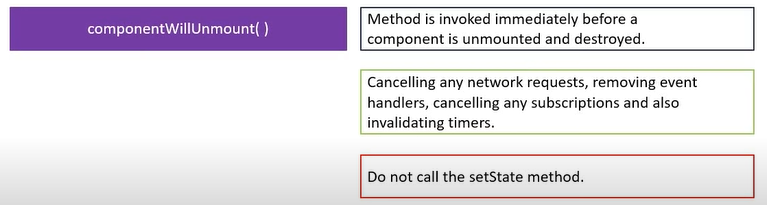




**Unmounting phase method**

**1🡪 componentWillUnmount():**

* This method is invoked immediately before a component is unmounted and destroyed.
* This method can be used to perform some cleanup task like cancelling any network requests, removing event handlers, cancelling any subscription and also invalidating timers.
* We should not call setState method here, because component is never rendered once it’s unmounted



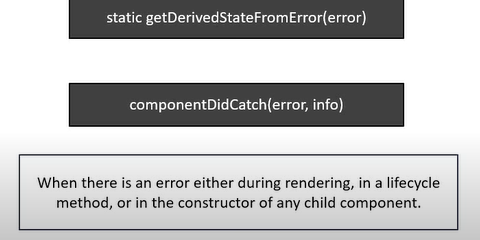
**Error handling Phase Method**

It has 2 methods,

**1🡪 static getDerivedStateFromError(error):**

**2🡪 componentDidCatch(error, info):**

Both these methods are called when there is error either during rendering in a lifecycle method, or in the constructor of any child component.



* When run time error occurs during rendering then entire application breaks i.e. react unmounts whole component tree.
* It’ll be better if we can catch the error and display a fallback UI. We can do this using error boundary
* A class component that implements either one or both of the error handling phase methods i.e. getDerivedStateFromError() or componentDidCatch() becomes error boundary.
* The static method getDerivedStateFromError() is used to render a fallback UI after an error is thrown and the componentDidCatch() method is used to log the error information.

import React from "react";

function Hero({ heroName }) {

  if (heroName == "Joker") {

    throw new Error("Not a hero");

  }

  return <div>{heroName}</div>;

}

export default Hero;

import React, { Component } from "react";

class ErrorBoundry extends Component {

  constructor(props) {

    super(props);

    this.state = {

      hasError: false,

    };

  }

  static getDerivedStateFromError(error) {

    return {

      hasError: true,

    };

  }

  componentDidCatch(error, info) {

    console.log(error);

    console.log(info);

  }

  render() {

    if (this.state.hasError) {

      return <h1>Something went wrong</h1>;

    }

    return this.props.children;

  }

}

export default ErrorBoundry;

import React from "react";

import "./App.css";

import Hero from "./components/Ch13p4Hero";

import ErrorBoundry from "./components/Ch13p4ErrorBoundry";

function App() {

  return (

    <div className="App">

      <ErrorBoundry>

        <Hero heroName="Batman" />

        <Hero heroName="Superman" />

        <Hero heroName="Joker" />

      </ErrorBoundry>

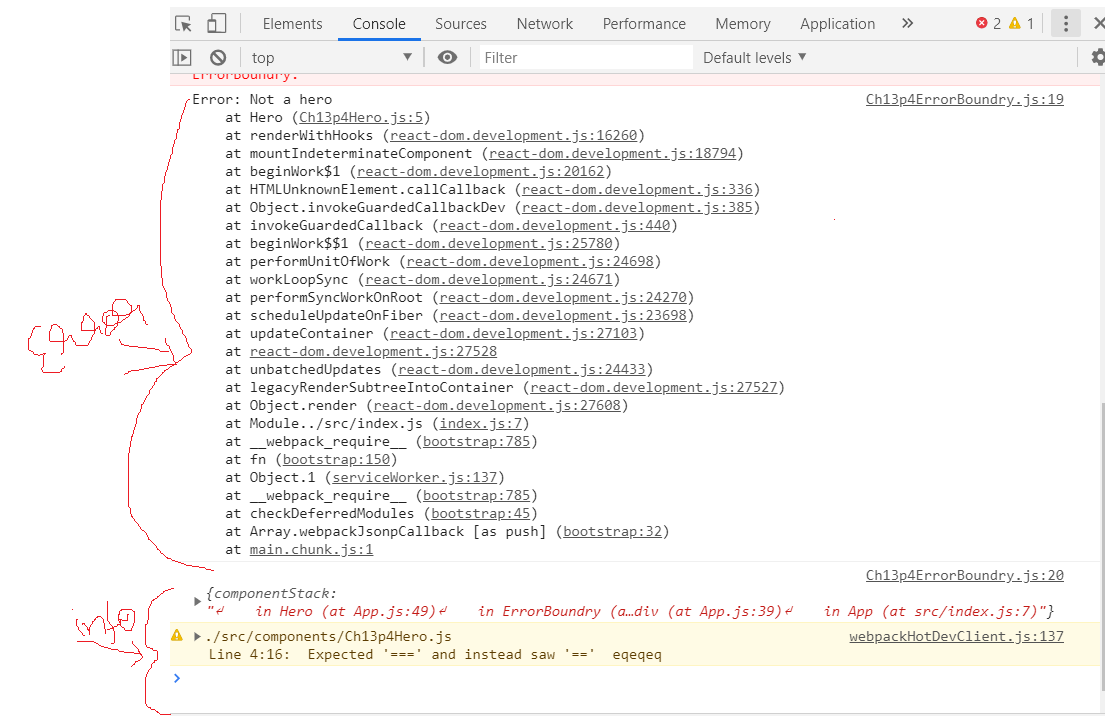
    </div>

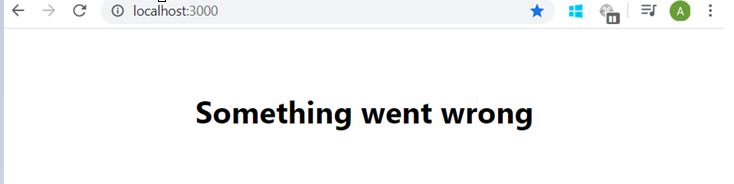
  );

}

export default App;







import React from "react";

import "./App.css";

import Hero from "./components/Ch13p4Hero";

import ErrorBoundry from "./components/Ch13p4ErrorBoundry";

function App() {

  return (

    <div className="App">

      <ErrorBoundry>

        <Hero heroName="Batman" />

      </ErrorBoundry>

      <ErrorBoundry>

        <Hero heroName="Superman" />

      </ErrorBoundry>

      <ErrorBoundry>

        <Hero heroName="Joker" />

      </ErrorBoundry>

    </div>

  );

}

export default App;



Note: error boundaries do not work in event handlers e.g. if you have click handler and u want to catch error in it then you have to use regular try catch block not error boundaries.

