Three phases of component lifecycle ->

1. Mounting
2. Updating
3. Unmounting

1. Mounting phase
   1. constructor()
   2. static getDerivedStateFromProps()
   3. render()
   4. componentDidMount()

   Legacy method -> componentWillMount()

2. Updating phase
   1. static getDerivedStateFromProps()
   2. shouldComponentUpdate()
   3. render()
   4. getSnapshotBeforeUpdate()
   5. componentDidUpdate()

3. Unmounting phase
   1. componentWillUnmount

4. Error Handling
   1. static getDerivedStateFromError()
   2. componentDidCatch()

---------- render() ----------

It is the only required method in a class comp. Others are optional.The render() function should be pure, meaning that it does not modify component state, it returns the same result each time it's invoked, and it does not directly interact with the browser.

If you need to interact with the browser, perform your work in componentDidMount() or the other lifecycle methods instead.

render() will not be invoked if shouldComponentUpdate() returns false.

Do not set state inside the render fn bcoz every time state changes, render() is called. If state is changed inside teh render(), it will bre called again. And agin the state will bw changed, though it will be not changed but the setState fn will run. It will again call render() fn. This way it will go into an infinite loop.

----------constructor() ----------

If you don't initialize state and you don't bind methods, you don't need to implement a constructor for your React component.

The constructor for a React component is called before it is mounted. When implementing the constructor for a React.Component subclass, you should call super(props) before any other statement. Otherwise, this.props will be undefined in the constructor, which can lead to bugs.

constructor() method is used for ->

- Initializing local state by assigning an object to this.state.
- Binding event handler methods to an instance.

You should not call setState() in the constructor(). Instead, if your component needs to use local state, assign the initial state to this.state directly in the constructor

```
constructor(props) {
    super(props);

    // Don't call this.setState() here!

    this.state = { counter: 0 };
    this.handleClick = this.handleClick.bind(this);
}
```

Constructor is the only place where you should assign this.state directly. In all other methods, you need to use this.setState() instead.

Avoid introducing any side-effects or subscriptions in the constructor. For those use cases, use componentDidMount() instead

---------- componentDidMount() ----------

componentDidMount() is invoked immediately after a component is mounted (inserted into the tree). Initialization that requires DOM nodes should go here. If you need to load data from a remote endpoint, this is a good place to instantiate the network request.

This method is a good place to set up any subscriptions. If you do that, don't forget to unsubscribe in componentWillUnmount().

You can call setState() inside the componentDidMount() immediately. It will trigger an extra rendering, but it will happen before the browser updates the screen. This guarantees that even though the render() will be called twice in this case, the user won't see the intermediate state. Use this pattern with caution because it often causes performance issues. In most cases, you should be able to assign the initial state in the constructor() instead. It can, however, be necessary for cases like modals and tooltips when you need to measure a DOM node before rendering something that depends on its size or position.

componentDidMount() will run at the last. The order is:

constructor > render> child constructor > child render > child componentDidMount() > componentDidMount().

---------- componentDidUpdate() ----------

componentDidUpdate(prevProps, prevState, snapshot)

componentDidUpdate() is invoked immediately after updating occurs. This method is not called for the initial render.

Use this as an opportunity to operate on the DOM when the component has been updated. This is also a good place to do network requests as long as you compare the current props to previous props (e.g. a network request may not be necessary if the props have not changed).

```
componentDidUpdate(prevProps) {
     // Typical usage (don't forget to compare props):
     if (this.props.userID !== prevProps.userID) {
       this.fetchData(this.props.userID);
     }
}
```

You may call setState() immediately in componentDidUpdate() but note that it must be wrapped in a condition like in the example above, or you'll cause an infinite loop. It would also

cause an extra re-rendering which, while not visible to the user, can affect the component performance.

If your component implements the getSnapshotBeforeUpdate() lifecycle (which is rare), the value it returns will be passed as a third “snapshot” parameter to componentDidUpdate(). Otherwise this parameter will be undefined.

componentDidUpdate() will not be invoked if shouldComponentUpdate() returns false.

---------- componentWillUnmount() ----------

componentWillUnmount() is invoked immediately before a component is unmounted and destroyed. Perform any necessary cleanup in this method, such as invalidating timers, canceling network requests, or cleaning up any subscriptions that were created in componentDidMount().

You should not call setState() in componentWillUnmount() because the component will never be re-rendered. Once a component instance is unmounted, it will never be mounted again.

RARELY USED LIFECYCLE METHODS

---------- shouldComponentUpdate() ----------

shouldComponentUpdate(nextProps, nextState)

Use shouldComponentUpdate() to let React know if a component’s output is not affected by the current change in state or props. The default behavior is to re-render on every state change, and in the vast majority of cases you should rely on the default behavior.

shouldComponentUpdate() is invoked before rendering when new props or state are being received. Defaults to true. This method is not called for the initial render or when forceUpdate() is used.

This method only exists as a performance optimization. Do not rely on it to “prevent” a rendering, as this can lead to bugs. Consider using the built-in PureComponent instead of writing shouldComponentUpdate() by hand. PureComponent performs a shallow comparison of props and state, and reduces the chance that you’ll skip a necessary update.

Currently, if shouldComponentUpdate() returns false, then UNSAFE_componentWillUpdate(), render(), and componentDidUpdate() will not be invoked.

---------- static getDerivedStateFromProps() ----------

static getDerivedStateFromProps(props, state)

getDerivedStateFromProps is invoked right before calling the render method, both on the initial mount and on subsequent updates. It should return an object to update the state, or null to update nothing.

This method exists for rare use cases where the state depends on changes in props over time. For example, it might be handy for implementing a <Transition> component that compares its previous and next children to decide which of them to animate in and out.

---------- getSnapshotBeforeUpdate() ----------

getSnapshotBeforeUpdate(prevProps, prevState)

getSnapshotBeforeUpdate() is invoked right before the most recently rendered output is committed to e.g. the DOM. It enables your component to capture some information from the DOM (e.g. scroll position) before it is potentially changed. Any value returned by this lifecycle method will be passed as a parameter to componentDidUpdate().

ERROR BOUNDARIES

Error boundaries are React components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of the component tree that crashed. Error boundaries catch errors during rendering, in lifecycle methods, and in constructors of the whole tree below them.

A class component becomes an error boundary if it defines either (or both) of the lifecycle methods static getDerivedStateFromError() or componentDidCatch(). Updating state from these lifecycles lets you capture an unhandled JavaScript error in the below tree and display a fallback UI.

Only use error boundaries for recovering from unexpected exceptions; don’t try to use them for control flow.

Error boundaries only catch errors in the components below them in the tree. An error boundary can’t catch an error within itself.

---------- static getDerivedStateFromError() ----------

static getDerivedStateFromError(error)

This lifecycle is invoked after an error has been thrown by a descendant component. It receives the error that was thrown as a parameter and should return a value to update state.

```
class ErrorBoundary extends React.Component {
  constructor(props) {
   super(props);
     this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
     // Update state so the next render will show the fallback UI.
     return { hasError: true };  }
          render() {
            if (this.state.hasError) {
            // You can render any custom fallback UI
            return <h1>Something went wrong.</h1>;
          }
      return this.props.children;
  }
}
```

getDerivedStateFromError() is called during the “render” phase, so side-effects are not permitted. For those use cases, use componentDidCatch() instead.

---------- componentDidCatch() ----------

componentDidCatch(error, info)

This lifecycle is invoked after an error has been thrown by a descendant component. It receives two parameters:

1. error - The error that was thrown.
2. info - An object with a componentStack key containing information about which component threw the error.

componentDidCatch() is called during the “commit” phase, so side-effects are permitted. It should be used for things like logging errors:

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // Update state so the next render will show the fallback UI.
    return { hasError: true };
  }

  componentDidCatch(error, info) {
      // Example "componentStack":
      //   in ComponentThatThrows (created by App)
      //   in ErrorBoundary (created by App)
      //   in div (created by App)
      //   in App
      logComponentStackToMyService(info.componentStack);
 }
  render() {
    if (this.state.hasError) {
      // You can render any custom fallback UI
      return <h1>Something went wrong.</h1>;
    }
    return this.props.children;
  }
}
```

In the event of an error, you can render a fallback UI with componentDidCatch() by calling setState, but this will be deprecated in a future release. Use static getDerivedStateFromError() to handle fallback rendering instead.