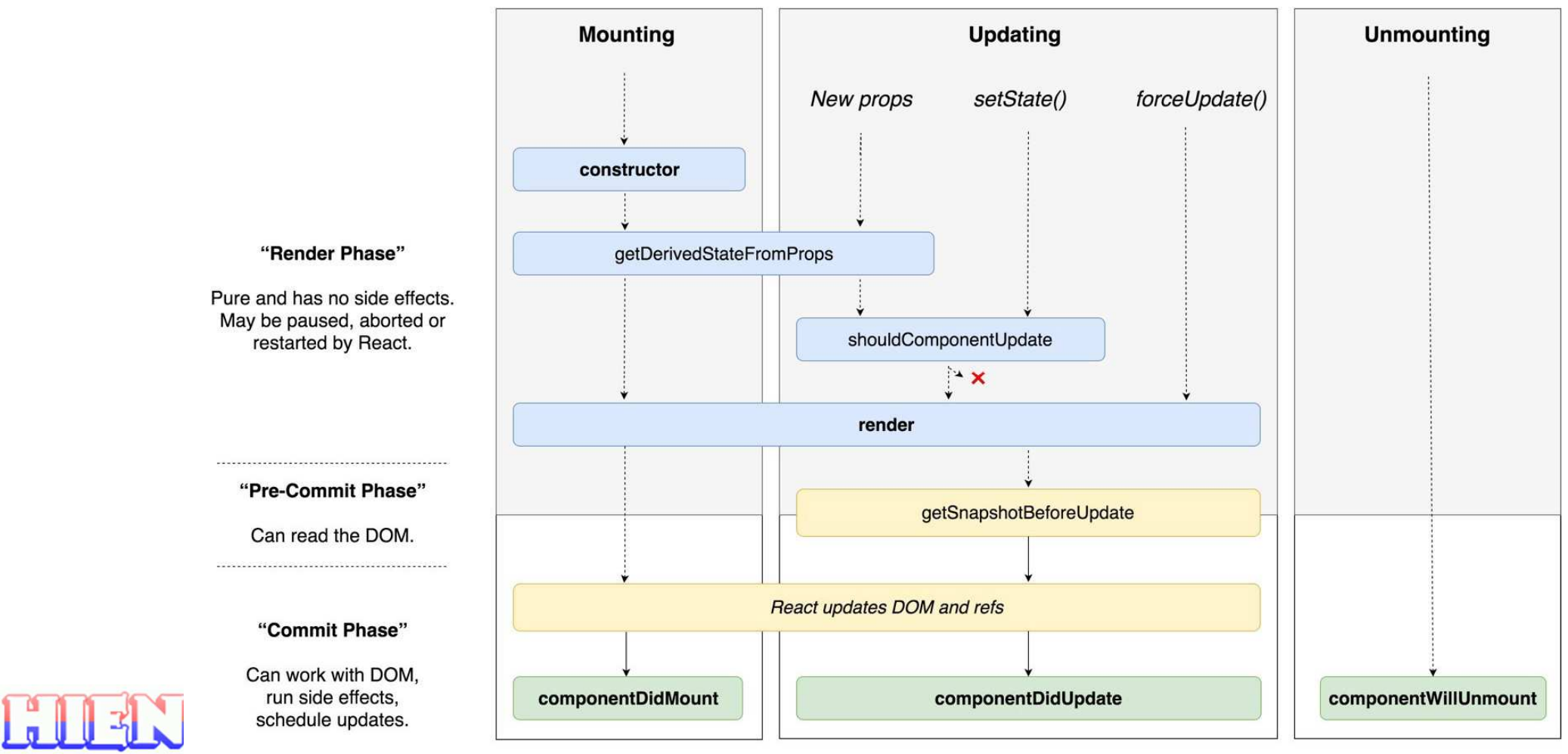
REACT NATIVE

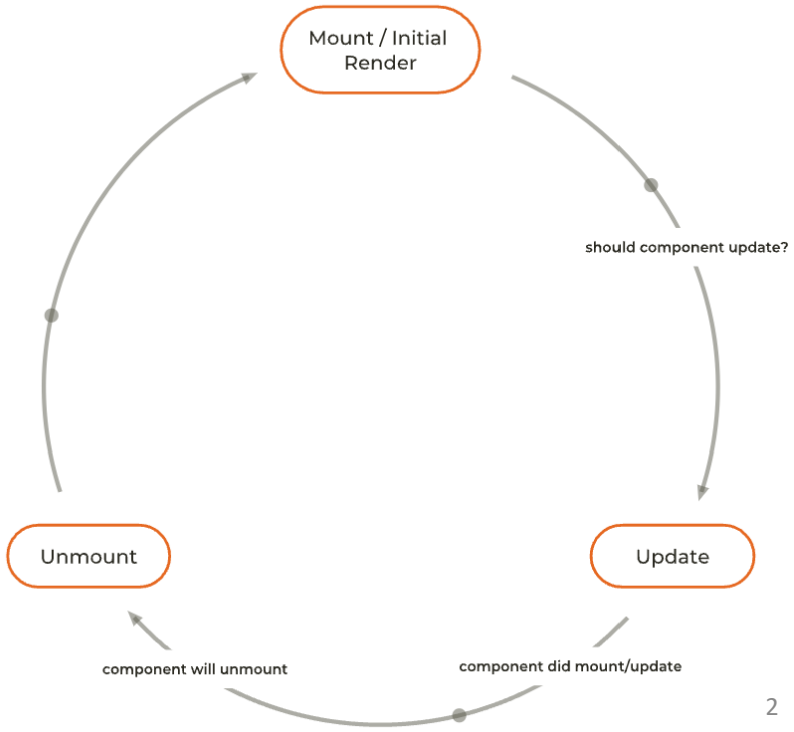
Component Lifecycle

Class component approach



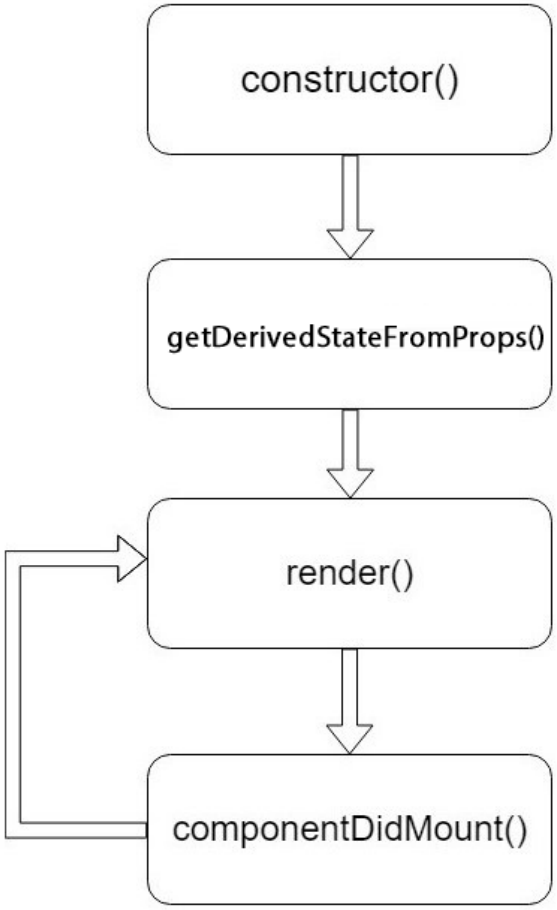
React Component Lifecycle

- Lifecycle of a component:
  - Initial Render or **Mount**
  - **Update** (When the states used in the component or props added to the component is changed)
  - **Unmount**



Mounting

- It will be in the following order
  1. constructor()
  2. static getDerivedStateFromProps()
  3. render()
  4. componentDidMount()



constructor(props)

- This method create a component, if not initializing the state or binding methods, does not need to declare this method

```
export default class Clicker extends Component {
    constructor(props) {
        super(props);
        this.handleClick = this.handleClick.bind(this);
        this.state = {
            clicks: 0
        };
    }

    handleClick() {
        this.setState({
            clicks: this.state.clicks + 1
        })
    }
    //...
}
```

constructor(props)

- Don’t transfer props to state! Handling logic will be very complicated later

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

    // DON'T DO THIS
    this.state = { color: props.color };
}
```

static getDerivedStateFromProps(props, state)

- This method 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 only one purpose. It enables a component to update its internal state as the result of **changes in props**

```
static getDerivedStateFromProps(props, state) {
    // Any time the current user changes,
    // Reset any parts of state that are tied to that user.
    if (props.userID !== state.prevPropsUserID) {
        return {
            email: props.defaultEmail,
            prevPropsUserID: props.userID
        };
    }
    return null;
}
```

render()

- This is the only required method when creating a component, which requires return one of the values below:
  - React element
  - Arrays and fragments
  - Portals
  - String and Numbers
  - Booleans or null
- This method will not be called if **shouldComponentUpdate()** return false

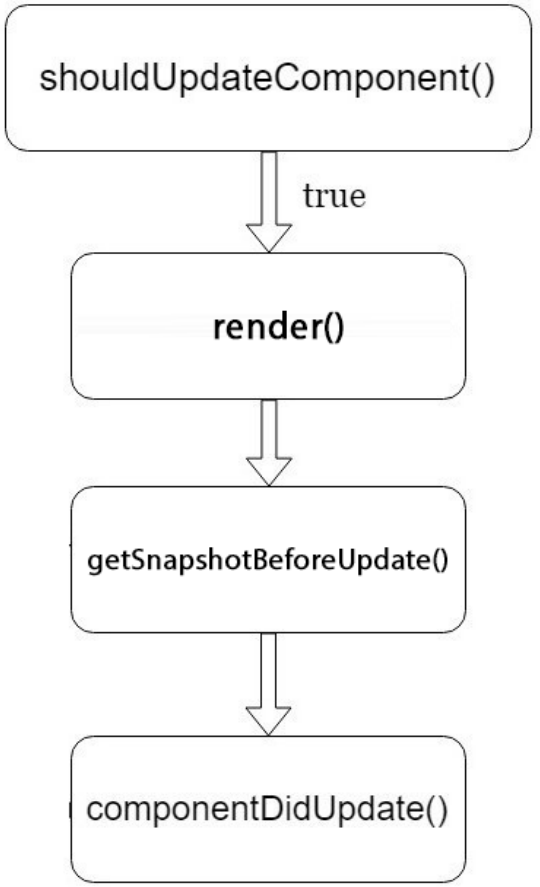
componentDidMount()

- Component has been rendered, it's time to call AJAX or setState

```
componentDidMount() {
    fetch('https://gitconnected.com')
        .then((res) => {
            this.setState({
                user: res.user
            });
        });
}
```

Updating

- These methods will be called when there is a change of state or props
  1. static getDerivedStateFromProps()
  2. shouldComponentUpdate()
  3. render()
  4. getSnapshotBeforeUpdate()
  5. componentDidUpdate()



shouldComponentUpdate(nextProps, nextState)

- Improve performance of React
- Is invoked before rendering when new props or state are being received
- Default value is true
- Not called for the initial render or when **forceUpdate()** is used

```
shouldComponentUpdate(nextProps, nextState) {
    return this.props.clicks !== nextProps.clicks;
}
```

getSnapshotBeforeUpdate()

- Is invoked right before rendered output is committed to the DOM
- It enables component to capture some information from the DOM (Ex: scroll position) before it is potentially changed
- Values return from this function will be passed as a parameter to **componentDidUpdate()**

componentDidUpdate(prevProps, prevState, snapshot)

- Is invoked immediately after updating occurs
- This method is not called for the initial render
- If call setState in this function, the conditional sentence must be included, otherwise it will be repeated infinitely
- If the method **getSnapshotBeforeUpdate()** is implemented, the return value will be include in snapshot parameter, otherwise undefined
- This function will not be called if **shouldComponentUpdate()** return false

Unmounting

- The method is called before removing the component from DOM
  - componentWillUnmout()
- This method can be use to remove listener, setInterval functions or cancel network request

```
componentWillUnmount() {
    window.removeEventListener('resize', this.resizeEventHandler);
}
```

React Native Hooks

useEffect

useEffect

- You can think of **useEffect Hook as componentDidMount, componentDidUpdate, and componentWillUnmount combined**.
- By default, **useEffect runs both after the first render *and* after every update**.
- For first render:

```
useEffect(() => {
  console.log("hello");
}, []);
```

- For every update:

```
useEffect(() => {
  console.log("hello");
}, [fieldChange]);
```

Ref: https://dev.to/fahadprod/react-hooks-how-to-use-usestate-and-useeffect-example-2h51

Anatomy of the useEffect hook

```
useEffect(() => {
    // Mounting

    return () => {
        // Cleanup function
    }
}, [//Updating])
```

useEffect(sideEffecFn, [dependencies]): cleanupFn

- **sideEfectFn**: A function that can perform a side effect, e.g. async call to fetch data
- **dependencies** (Array): A list of values that if changed will trigger the sideEffectFn and cause a re-render
- **cleanupFn**: the (optional) returned value of the side effect - triggered before each re-render - used for cleaning up, e.g. unregistering from events

useEffect

```
const Counter = props => {
  const [count, setCount] = useState(0);
  useEffect(() => {
    console.log(`The count is now ${count}`)
  });
  return (
    <button onClick={() => setCount(count + 1)}>
      { count }
    </button>
  )
}
```

```
The count is now 0
--- Button onClick ---
cleaning up
The count is now 1
--- Button onClick ---
cleaning up
The count is now 2
--- Button onClick ---
cleaning up
The count is now 3
```

The dependencies optional parameter

- Every time the component renders the effect is ‘useEfect’ is triggered.
- It compares the values in this array to the values of the previous execution.
- If the values do not change then the sideEffectFn will not re-execute.
  - No array given (undefined) - will execute on every render (componentDidMount + componentDidUpdate)
  - An empty array given - will execute once (componentDidMount)
  - Array with values - will execute only if one of the values has changed

useEffect Summary

```
useEffect(() => {
    // EVERY
    // No dependencies defined
    // Always execute after every render
    return () => {
        // Execute before the next effect or unmount.
    };
});
useEffect(() => {
    // ONCE
    // Empty dependencies
    // Only execute once after the FIRST RENDER
    return () => {
        // Execute once when unmount
    };
}, []);
```

useEffect Summary (cont.)

```
useEffect(() => {
    // On demand
    // Has dependencies
    // Only execute after the first RENDER or filters state changes
    return () => {
        // Execute before the next effect or unmount.
    };
}, [filters]);
```

Some key points

- useEffect is effect which is equal to componentDidMount and ComponentDidUpdate.
- which means useEffect will run on mount and update
- will run only after the DOM is applied or DOM mutation is done.
- every time whenever a local state is changed this effect will run.
- we can make our effect run depending on other state by passing it in second argument as array
- we can make effect run only once by passing empty array in second argument.

Error Handling

- Regardless of where the error is in component, it will call this method
  - componentDidCatch()
- This function will handle error when a component fails, and it will show the error on UI

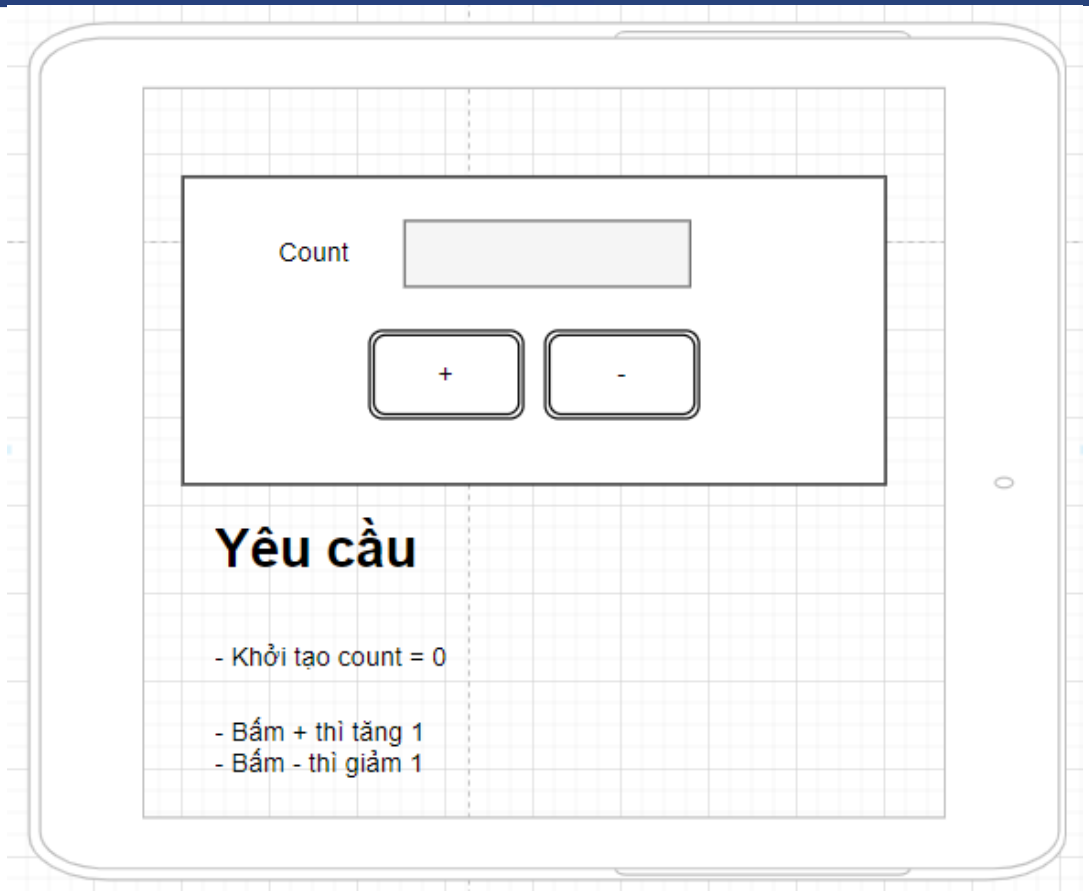
```
export default class ErrorBoundary extends React.Component {
    state = { hasError: false };

    componentDidCatch() {
        this.setState( { hasError: true });
    }

    render() {
        if (this.state.hasError) {
            return <Text>Error in Component</Text>;
        }
        return this.props.children;
    }
}
```

Exercise

- Design a GUI that permit user can increment and decrement the value.

- Hint: Using

useState, useEffect



Thank you!