Interview

- React Related
  - React
    - Virtual DOM vs Real DOM

Virtual DOM	“Real” DOM
Can’t directly update HTML	Directly updates and manipulates HTML
Acts as a copy of the real DOM, which can be frequently manipulated and updated without a page refresh	Creates a new DOM/full repaint if it is updated
More of a pattern than a specific technology	An object-based representation of an HTML document + an interface for manipulating that object
Synced with the real DOM with ‘react-dom’	

      - The key is diffing
        - virtual dom looks at one state vs the next state (diffing)
        - and update just that individual parts

        When the VDOM gets updated, React compares it to to a previous *snapshot* of the VDOM and then *only* updates what has changed in the real DOM. If nothing changed, the real DOM wouldn't be updated at all. **This process of comparing the old VDOM with the new one is called *diffing***.

        Real DOM updates are slow because they cause an actual re-draw of the UI. React makes this more efficient by updating the smallest

amount possible in the real DOM.

▼ re-rendering

- parent re-render
- children → lift up state to parent → all the parent's children will re-render
- key change
- prop changes
- internal state change

!!! array and object as state, MUST give a new object, else it is the same reference

React only does a shallow comparison, need a new reference to cause re-render

▼ state change

Changing the **state** means that React triggers an update when we call the `setState` function (in React hooks, you would use `useState`). This doesn't only mean the component's render function will be called, but also that **all its subsequent child components will re-render, regardless of whether their props have changed or not**.

▼ ways to prevent unnecessary updates

React.memo: prevents your React Hook components from rendering when the props don't change.

shouldComponentUpdate: This function is one of React's lifecycle functions and allows us to optimize rendering performance by telling React when to update a class component.

Set the key attribute: In some cases, React relies on the `key` attribute for **identifying components and optimizing performance**.

▼ shadowDOM

        - shadowDOM is broswer specific, like slider, playbutton, pause button(web component)
            - CSS in shadowDOM will not leak out to DOM

▼ React Limitation

- It is a library, not a framework
- It is largee

▼ JSX

- write JavaScript with an HTML-like syntax
- produces elements that represent objects
- JSX in React =React.createElement()

    ▼ Element vs Component

        - element: produced by JSX that represent an object
            - just
        - component: a function that **returns** a element
            - add logic then return

▼ props

    ▼ Passing value from parents to child?

    props

    ▼ Passing value from child to parent?

        1. passing a function prop to a child
        2. child calls the function

    ▼ Prop-drilling

        1. passing props multiple level

    ▼ Can you modify props

no, props are read-only

▼ state && lifecycle

▼ The different between state vs props

state is scoped to that function

state cannot be access outside of that component unless pass as props

▼ The different between state in class component vs state in functional component

state in class component: persist like object

state in functional component: recall as functional component is recalled

▼ Class Component

▼ Component Life Cycle

executing code in different sequence

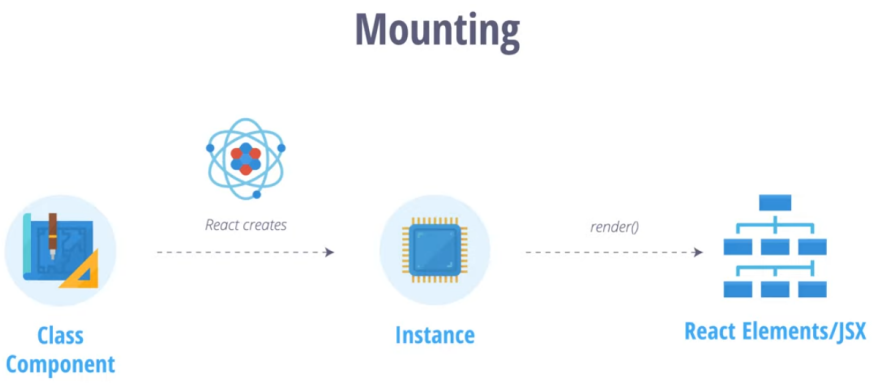
- constructor is before first render
- mounting
  - render,componentDidMount, after first render
- updating
  - render, componentDidUpdate
- unmouting
  - componentWillUnmount, cancel subsribtion, remove event listeners

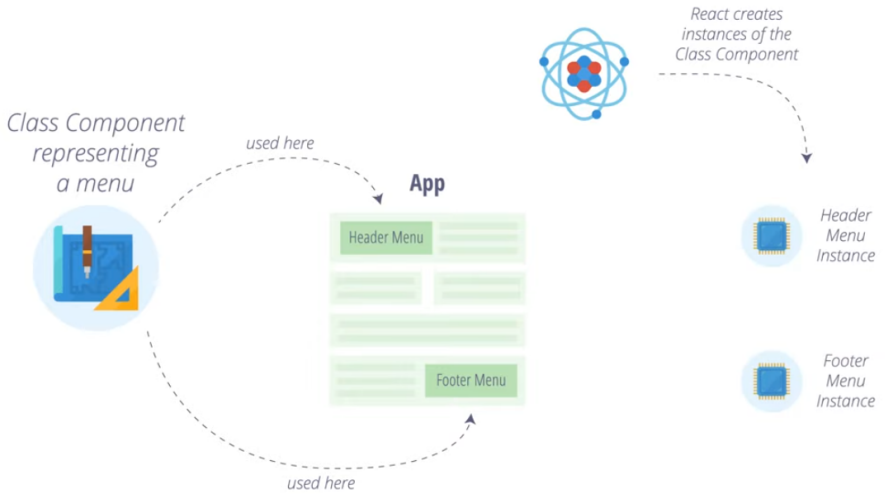
```
import { Component } from 'react'
export class MyClassComponent extends Component {
  constructor() {
    super()
    this.state = {
      counterValue: 0,
```

```
    }
  }
  render() {
    return (
      <div>
        <button
          onClick={() =>
            this.setState((currentState) => {
              return {
                counterValue: currentState + 1,
              }
            })
          }
        >
          Click Me
        </button>
        {this.props.text}
      </div>
    )
  }
}
```

▼ mounting

- React creates a instance of this class, then calls render method, then return JSX element(Mounting)
- can only have one state object
- before hooks, only can use state in class component





unmount: instance is throwed away

```
interface AppSate{
counter:number;
}
interface AppProps{
color?:string;
}

class App extends React.Component<AppProps,AppSate>{
state={counter:0)}
}
==============================================
or
==============================================
class App extends React.Component<AppProps,AppSate>{
constructor(props:AppProps){
super(props)
this.state={counter:0}
}

}
```

▼ Functional Component (new)

▼ reuse custom hooks

```
import { useState } from 'react'
export function MyFunctionalComponent(props) {
  const [counterValue, setCounterValue] = useState(0)
  return (
```

```
    <div>
      <button
        onClick={() =>
          setCounterValue((currentCounterValue) => currentCounterValue +
1)
        }
      >
        Click Me
      </button>
      {counterValue}
    </div>
  )
}
```

- ▼ useEffect

  componentDidMount(),componentDidUpdate()&componentWillUnmount combined

  - ▼ side-effects with useEffects (lifecycle)
    - [] → Runs on mount, once
    - [variable]→ Runs on mount and when variable changes
    - No array → Runs on mount and on every state change

  return value is the cleanup function: eventListener, fetch cancellation

- ▼ Functional Component (old)
  - return a jsx Element
  - no side-effects
    - no componentDidMount, componentDidUpdate
  - no state
    - no persisted data
    - every render, the body of functional component is run again
    - no accessing the previous data

- ▼ Refs
  - ▼ What is the difference between refs and state variables?
    - refs have value can persist between renders

- refs do not trigger re-renders

▼ Best time to use Refs

- Managing focus or media playback, text selection
- Triggering animations
- Integrating with DOM libraries (give third party library ability to grab DOM)

▼ What is the proper way to update a ref in functional component?

- useEffect

```
const ref=React.useRef(null)

React.useEffect(
()=>{
ref.current.focus()
},[])

return
<input ref={ref}/>
<button onClick={()=>ref.current.focus()}>Focus</button>
...
```

▼ Context

▼ What is the difference between the Context API and prop drilling?

- prop drilling: specific child can use the props "username,bio"
- Define Context at top level, define value at the top level. Anything inside on the component tree can use the values "useAuth"

▼ When should not you use the Context API?

- CAN lead to unnecessary re-render
- Can create more Context API for specific component
- when things need to be shared across app tree
- Less unnecessary values at top level context.
- should be at top level: IsAuth

▼ memo&&useMemo && useCallback

▼ memo

the result of the function wrapped in React.memo is saved in memory and **returns the cached result** if it's being called with the **same input** again.

```
============================================================
// !!Count will re-render everytime its parent re-renders
<div><input value={text} onChange={e=>setText(e.target.value)}/>
<Count/>
</div>

//prevet re-rencer
============================================================
// React.memo prevents re-render
export const Count=React.memo( ()={
const [count,setCount]=React.useState(0)
const renders=React.useRef(0)
return (
<div>
<div>Count: {count}</div>
</div>
)
})

//function breaks memo
============================================================
// passing a function will break React.memo
// since every re-render will generate a new function
// shallow comparison
<div><input value={text} onChange={e=>setText(e.target.value)}/>
<Count aFunction={someFunction}/>
</div>

------------------------------------------------------------
//memorize a func, only change it when [dep] changes
const someFunction=React.useCallback(()=>{....},[dep])
<div><input value={text} onChange={e=>setText(e.target.value)}/>
<Count aFunction={someFunction}/>
</div>

//object breaks memo
============================================================
// passing a object will break React.memo
// since every re-render will generate a new function
```

```
// shallow comparison
cosnt data={isAuth:false}
<div><input value={text} onChange={e=>setText(e.target.value)}/>
<Count data={data}/>
</div>

-----------------------------------------------------------
//solution 1, compare prevProps and nextProps in child
export const Count=React.memo(({data})={
const [count,setCount]=React.useState(0)
const renders=React.useRef(0)
return (
<div>
<div>Count: {count}</div>
</div>
)
},(prevProps,nextProps)=>{
if(prevProps.data.isAuth!==nextProps.data.isAuth){
return false
}
return true
})

-----------------------------------------------------------
//solution 2, memorize object in parent
//new data object if [dep] changes
//data object never change with [] empty dep
cosnt data=React.useMemo(()=>({isAuth:false}),[dep])
<div><input value={text} onChange={e=>setText(e.target.value)}/>
<Count data={data}/>
</div>
```

▼ Danger\Pitfall

▼ **Dealing with objects**

it only does a **shallow comparison** of the component's properties. The `data` variable is being re-declared on every update of `App`. This leads to the objects **not actually being the same** because they have **different references**.

▼ **Solving this with areEqual**

React.memo provides a solution for this in its second parameter. This parameter accepts a second `areEqual` function, which we can use to control when the component should update.

```
const TileMemo = React.memo(() => {
  let updates = React.useRef(0);
  return (
    <div className="black-tile">
      <Updates updates={updates.current++} />
    </div>
  );
}, (prevProps, nextProps) => {  if (prevProps.data.test ===
nextProps.data.test) {    return true; // props are equal
 }  return false; // props are not equal -> update the comp
onent});
```

▼ useMemo()

you can wrap the object in `React.useMemo()`, which will memoize the variable and not create a new object.

```
const data = React.useMemo(() => ({
  test: 'data',
}), []);
//The second parameter of useMemo is an array with the
 dependencies of the variable. If one of them changes,
 React will recompute the value.

//In our case, this won't happen since the array is emp
ty.
```

▼ useCallback

Problem: functions behave just like objects, which leads to the same problem we had before. The `onClick` function is being declared each time `App` updates. `TileMemo` then thinks that `onClick` changed because the reference has changed.

```
const App = () => {
  const updates = React.useRef(0);
  const [text, setText] = React.useState('');
  const onClick = () => {    console.log('click');  };
  return (
    <div className="app">
      <div className="blue-wrapper">
        <input
          value={text}
```

```
          placeholder="Write something"
          onChange={(e) => setText(e.target.value)}
        />
        <Updates updates={updates.current++} />
        <Tile />
        <TileMemo onClick={onClick} />      </div>
    </div>
  );
};
```

We can solve this just like we did with objects, by memoizing the function.

```
const onClick = React.useCallback(() => {
  console.log('click');
}, []);
//Here the second parameter is again an array with dependencies which wi
ll trigger computation of the value if they change.
```

▼ **Why not use React.memo by default?**

- more memory consumption
- the component's props change frequently

▼ Lazy initialization of state

- initialState() is called on every re-render

```
function initialState() {
  console.log('expensive calculation')
  return 0
}

export default function App() {
  const [state, setState] = useState(initialState())
  return <div></div>
}
```

- lazy initialization

```
function initialState() {
  console.log('expensive calculation')
  return 0
```

```
}

export default function App() {
  const [state, setState] = useState(()=>initialState())
  return <div></div>
}
```

▼ useEffect vs useLayoutEffect

- useEffect

  1. The user performs an action, i.e., clicks the button.

  2. React updates the count state variable internally.

  3. React handles the DOM mutation.

  4. The browser paints this DOM change to the browser's screen.

  5. Only after the browser has painted the DOM change(s) is the `useEffect` function fired.

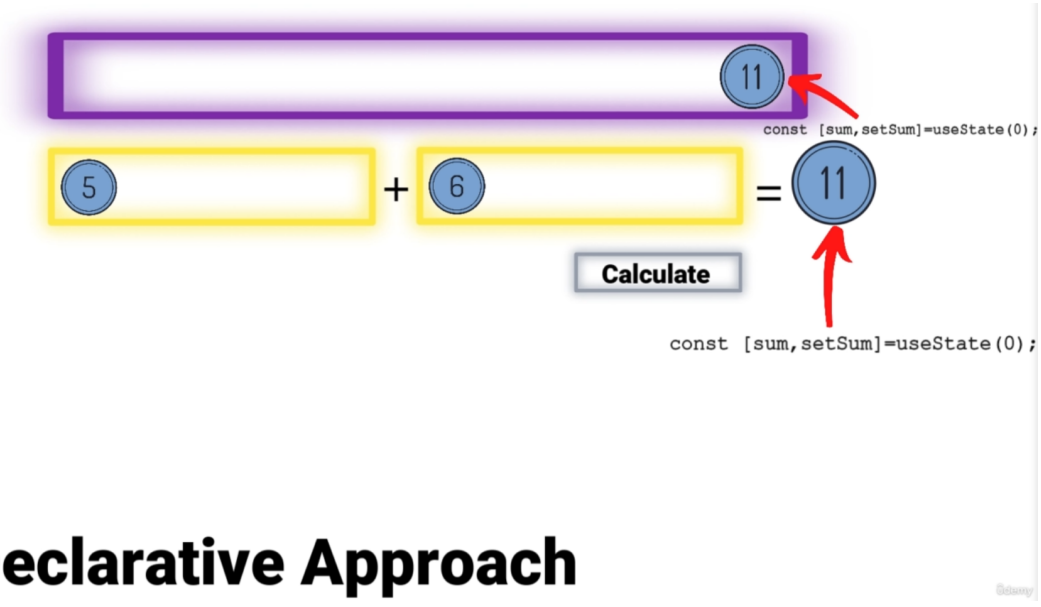
  !!!!What to note here is that the function passed to `useEffect` will be fired *only* after the DOM changes are painted to the screen.

- useLayoutEffect

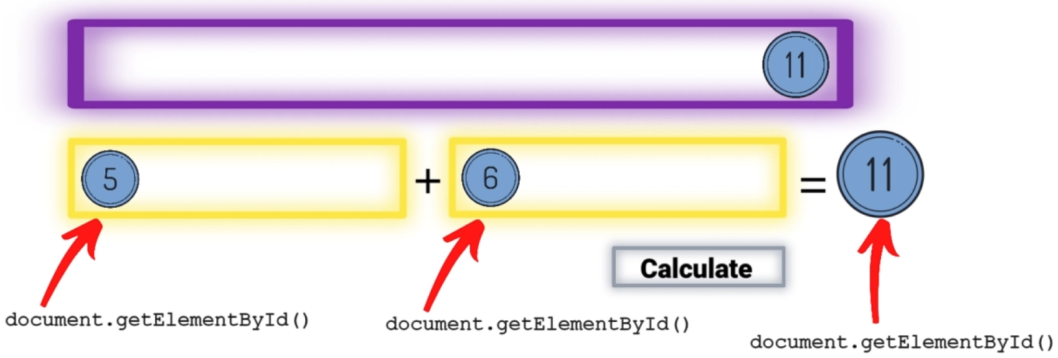
  1. `useLayoutEffect` doesn't really care whether the browser has painted the DOM changes or not. It triggers the function right after the DOM mutations are computed.
  2. If you rely on these refs to perform an animation as soon as the component mounts, then you'll find an unpleasant flickering of browser paints happen before your animation kicks in. This is the case with `useEffect`, but not `useLayoutEffect`.
  3. This can be useful if you need to make DOM measurements (like getting the scroll position or other styles for an element) and then make DOM mutations **or** trigger a synchronous re-render by updating state.

▼ Miscellaneouse

- ▼ When to use class component instead of functional component?
  - Errorboundry
- ▼ What is a higher order component?
  - HOC: takes in a component and return a new component
    - <Appshell>
- ▼ What is a portal?
  - UI library
  - render a children in DOM node
  - can live in anywhere in the DOM tree
  - for example: modal
- ▼ What are uncontrolled components?
  - like input user control and React does not control
- ▼ Declarative vs Imperative
  - ▼ Declarative :"clean my desktop"



  - ▼ Imperative:"1.take a towel, 2.wash it, 3.wipe..."



Imperative Approach

▼ state vs variable

state: like variable but React will watch the change, any change will cause re-render

▼ List & Map

```
let arr=[1,2,3,4,]
let nArr = arr.map((element, index) => element * index)
```

▼ Key

React needs a unique value to refer the element when making changes

Need id or other stable identity, NOT index of the map

▼ react-router

useLocation & query string parameter

```
import {useLocation} from "react-router-dom"
....
//url: localhostL3000/haha?price=$20&name=hui+bai
function useQuery(){
```

```
return new URLSearchParams(useLocation().search())
}
const query=useQuery()

return <div>{query.price}</div>
```

▼ redux

```
store:
global state object with multiple properties
==================================================
action:
dispatch action-type and data, what properties within store to change and change
 to what
==================================================
reducers:
switch base on action-type and define ways to manipulate store
==================================================
```

▼ General Interview

- CS fundamentals
- Problem solving strategy
    - hope for the best, prepare for the worst
- Self Confidence
- Consistent Practice

"I was the answer to their problem"