Class_Components

Most asked in Interviews since Older companies uses Class Components in their legacy Code.

Increases understanding about Components LifeCycle.

You will get to know how painful it was earlier and how modern solutions makes it easy.

Creating a class component

- import `React` or Destructure it `{Component} from "react"`
- Create a `class` extending `React.Component`
- create a function `render()` inside the `class` which returns a `jsx`

Passing props to Class Components

`Constructor` of the class components receive the props via `super` keyword

```
import React from 'react';

class Class_Component extends React.Component{
    constructor(props){
    super(props)
    }

}

//or

import {Component} from "react"
class Class_Component extends Component{

}
```

To use the props we need to use `this`

```
{this.props.value}

//Destuructre
const {value1, value2} = this.props
```

> **Why to use `super(props)` and `this`**
>
> - In JavaScript's class inheritance, the `Parent` constructor is responsible for:
>   - Allocating & initializing the instance (`this`)
>   - Setting up the internal prototype chain & properties
> - Until `super()` is called, `this` literally **doesn't exist yet** in the context of the child constructor.
> - `super()` calls the parent constructor → which creates & initializes `this`
> - React uses this to setup the `this` , `props` and internal stuff (state handling, updater, etc.) for the component instance.

States in Class Components

States are created inside the `constructor` since it's the first thing called when he component loads on the screen.

`State` is an object in Class Component.

```
cosntructor(props){

this.state{
    state_variable_1: initial_value,
    state_variable_2: initial_value,
}

render(){
    return(
    <p>Use State Like: {this.state.state_variable} </p>
)
}
}
```

Updating State Variables

> Never update class state variables **Directly** as
>
> ```
> this.state.state_variable = this.state.state_variable+1
> ```
>
> It won't Work

Correct way is to use `this.setState()` takes a object will contain the updated values of all the state_variables.

```
this.setState({
    state_variable_1 : new_value,
    // To use exisitng value of state_variable in the new_value
    state_variable_2 : this.state.state_variable_2 + 1
})
```

`this.setState` will only the update the states mentioned in it's object.

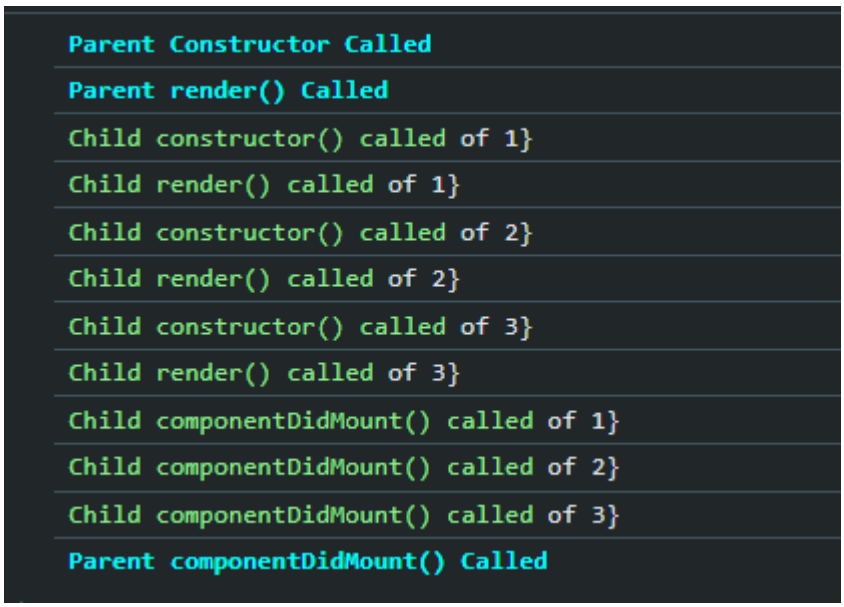
Life Cycle of Class_Components

When a Class Component is instantiated first `constructor()` is called and then `render()`

React provides a method `componentDidMount()` which is called after the component has mounted. (after render)

`componentDidMount()` is used to make `API` calls.

> Because Once we mount the component it is easer for us to place `API Content` and React will update the component blazing fast.
>
> If we wait until content is fetched and then mount the component it leads bad UX since we never know the response time taken to get the content.
>
> hence we make api_call later and until then mount the component as Skelton/Shimmer.
>
> **Remember: Render Quickly then fill the data.**



> Notice the `componentDidMount()` of child's are called at last together in respective order rather than after the child `render()` method.
>
> To understand this Lifecycle of Components

Lifecycle of Components

Refer the diagram: React lifecycle methods diagram

Component is Mounted in 2 phases:

1. **Render Phase** `contructor()` and `render()` is called. and this pure and the `DOM` is updated.
2. **Commit phase** : `componentDidMount()` is called in this phase. Side-effects can work in this phase since DOM is updated.

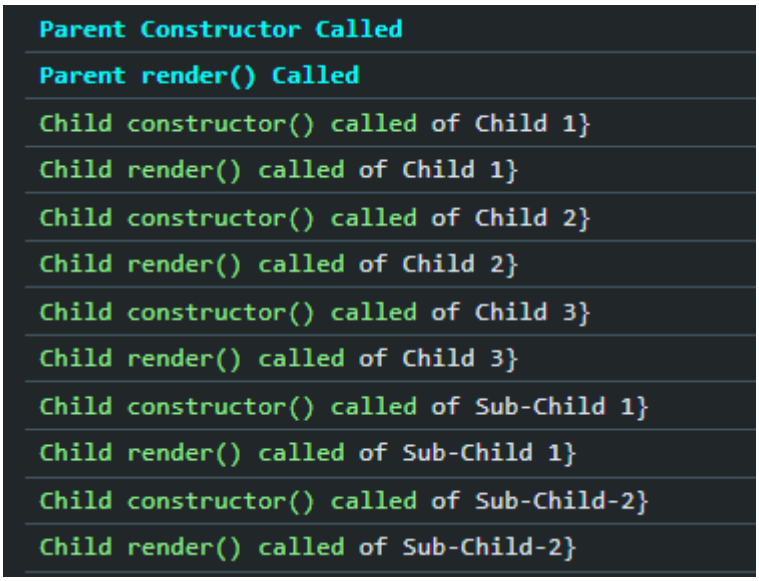
React optimizes by Batching all the `render phase` of child's. hence all are rendered then it batches `commit phase` of all the children.

> **Why React Batches Phases?**
>
> `DOM` manipulation is Expensive.
>
> - In `render phase` Virtual DOM are compared.
> - In `commit phase` changes are made to Real DOM
>
> React batches these phases to make the process faster. If these phases are executed individually for every child then websites becomes slow since DOM operations are computationally expensive.

```
Child componentDidMount() called of Child 1}
Child componentDidMount() called of Child 2}
Child componentDidMount() called of Sub-Child 1}
Child componentDidMount() called of Sub-Child-2}
Child componentDidMount() called of Child 3}
Parent componentDidMount() Called
```

> Why did the `compoentDidMount()` of sub children executed first?
>
> React doesn't exactly "batch by siblings."
> It does a **depth-first traversal**:
>
> - For each branch: finishes mounting all nested children first (including calling their `componentDidMount`),
> - Then goes to the next sibling
>
> In our Case for Commit Phase
>
> - React Encounters `Child 3` → then finds it has children:
>   - Renders `Sub-Child 1` → calls its `componentDidMount`
>   - Renders `Sub-Child-2` → calls its `componentDidMount`
> - Then finally calls `componentDidMount` for `Child 3`

In `render phase` React goes down upto last child of the parent. Eg:



When Encountered `Child-3` React sees it has child so goes to them as well

> Remember
>
> During the render phase → React discovers children *only when rendering the parent* .
> During the commit phase → React knows the full tree → can do proper DFS

Understanding Traversal

Traversal of Components depends upon the `Real-DOM`

`constructor() and render()` -> DFS

While these are called `Real-DOM` is not build yet, React First Executes the parent then realizes that child components exist and then it executes them.

> parent->child-1->child-2->sub_child-> child-3

`compoennetDidMount()` DFS , bottoms Up for Children.

At each node:

- React first goes into all its children (from left to right, recursively)
- After visiting and finishing all children, it calls the node's own `componentDidMount`

> Child-1->Sub-child-2 -> Child-2>Child-3->parent

> **Why Parent is at last**
>
> React's DFS traversal for `componentDidMount`:
>
> 1. Start at Parent
> 2. Go to first child: **Child 1**
>   - Child 1 has no children → immediately call `componentDidMount` of Child 1
> 3. Back to Parent, next child: **Child 2**
>   - Child 2 has a child → go deeper into Sub-Child 1 of Child-2
>   - Sub-Child has no children → call `componentDidMount` of Sub-Child 1 of Child-2
>   - Back to Child 2 → now call `componentDidMount` of Child 2
> 4. Back to Parent, next child: **Child 3**
>   - Child 3 has no children → call `componentDidMount` of Child 3
> 5. Finally, after finishing all children → call `componentDidMount` of Parent
>
> This is known as DFS Bottoms Up since traversal goes from bottom(Children) to
>
> up(parent)

React guarantees:

> "By the time a parent's `componentDidMount` runs, **not only is the real DOM for the children in place**, but also **children's own lifecycle side-effects (`componentDidMount`) have already been run**."

React intentionally calls children's `componentDidMount` before parent's

- to ensure side-effects, refs, subscriptions, measurements, etc., that happen in `componentDidMount of the children` are already done before the `parent's componentDidMount` runs.

`componentWillUnmount()`

React removes the subtree in one go, but:

- Calls `componentWillUnmount` on parent first, to let parent clean up its effects

- Then on each child, to clean up their effects

> - By the time `componentWillUnmount` is called, React **hasn't yet destroyed child components**
> - So the parent can still theoretically query child DOM if it wants.
> - parent removes higher-level resources, then children clean up their local resources.

Final Traversal:

```
Parent Constructor Called
Parent render() Called
Child constructor() called of Child 1}
Child render() called of Child 1}
Child constructor() called of Child 2}
Child render() called of Child 2}
Child constructor() called of Sub-Child 1 of Child-2}
Child render() called of Sub-Child 1 of Child-2}
Child constructor() called of Child 3}
Child render() called of Child 3}
Child componentDidMount() called of Child 1}
Child componentDidMount() called of Sub-Child 1 of Child-2}
Child componentDidMount() called of Child 2}
Child componentDidMount() called of Child 3}
Parent componentDidMount() Called
Parent componentWillUnMount() Called
Child componentWillUnMount() called of Child 1}
Child componentWillUnMount() called of Child 2}
Child componentWillUnMount() called of Sub-Child 1 of Child-2}
Child componentWillUnMount() called of Child 3}
```