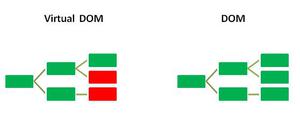
**What special about React?**

* React.js is declarative
* React.js is simple
* React.js is component based
* React.js supports server side
* React.js is extensive
* React.js is fast
* React.js is easy to learn
* reusable UI components.
* React is Flexible
  + React is remarkably flexible. Once you have learned it, you can use it on a vast variety of platforms to build quality user interfaces
* React Native

**What are features of ReactJs ?**

* JSX (JavaScript Syntax Extension)
* Virtual DOM
* One-way data binding
* Performance
* Server-side rendering
* React native

1. **JSX (JavaScript Syntax Extension):**[JSX](https://www.geeksforgeeks.org/reactjs-introduction-jsx/) is a combination of HTML and JavaScript. You can embed JavaScript objects inside the HTML elements. JSX is not supported by the browsers, as a result [Babel compile](https://www.geeksforgeeks.org/reactjs-using-babel/)r trans compile the code into JavaScript code.
2. **2.** **Virtual DOM:**DOM stands for [Document Object Model](https://www.geeksforgeeks.org/dom-document-object-model/). It is the most important part of the web as it divides into modules and executes the code. Usually, JavaScript Frameworks updates the whole DOM at once, which makes the web application slow. But react uses virtual DOM which is an exact copy of real DOM. Whenever there is a modification in the web application, the whole virtual DOM is updated first and finds the difference between real DOM and Virtual DOM. Once it finds the difference, then DOM updates only the part that has changed recently and everything remains the same.



In the above-shown figure, when the whole virtual DOM has updated there is a change in the child components. So, now DOM finds the difference and updates only the changed part.

**How Virtual DOM updates real DOM?**

Consider a virtual DOM object as a blueprint of the real DOM object. Whenever a JSX element gets rendered, every virtual DOM object gets updated.

\*\*Note- One may think updating every virtual DOM object might be inefficient, but that’s not the case. Updating the virtual DOM is much faster than updating the real DOM since we are just updating the blueprint of the real DOM.

React uses **two virtual DOMs** to render the user interface. One of them is used to store the current state of the objects and the other to store the previous state of the objects. Whenever the virtual DOM gets updated, react compares the two virtual DOMs and gets to know about which virtual DOM objects were updated. After knowing which objects were updated, react renders only those objects inside the real DOM instead of rendering the complete real DOM. This way, with the use of virtual DOM, react solves the problem of inefficient updating.

The virtual DOM (VDOM) is a programming concept where an ideal, or “virtual”, representation of a UI is kept in memory and synced with the “real” DOM by a library such as ReactDOM. This process is called [**reconciliation**](https://reactjs.org/docs/reconciliation.html).

### **The Diffing Algorithm**

## When diffing two trees, React first compares the two root elements. The behaviour is different depending on the types of the root elements.

### **Is the Shadow DOM the same as the Virtual DOM?**

### No, they are different. The Shadow DOM is a browser technology designed primarily for scoping variables and CSS in web components. The virtual DOM is a concept implemented by libraries in JavaScript on top of browser APIs.

### **What is “React Fiber”?**

## Fiber is the new reconciliation engine in React 16. Its main goal is to enable incremental rendering of the virtual DOM.

**One-way data binding**

One-way data binding, the name itself says that it is a one-direction flow. The data in react flows only in one direction i.e., the data is transferred from top to bottom i.e., from parent components to child components. The properties(props) in the child component cannot return the data to its parent component but it can have communication with the parent components to modify the states according to the provided inputs. This is the working process of one-way data binding. This keeps everything modular and fast.

### **What are the differences between functional and class components?**

**Declaration**

Functional components are nothing but JavaScript functions and therefore can be declared using an arrow function or the function keyword:

Class components, on the other hand, are declared using the ES6 class

**Handling props**

In functional components, the handling of props is pretty straightforward. Any prop provided as an argument to a functional component can be directly used inside HTML elements:

In the case of class components, props are handled in a different way:

class StudentInfo extends React.Component{

constructor(props){

super(props);

}

render(){

return(

<div className="main">

<h2>{this.props.name}</h2>

<h4>{this.props.rollNumber}</h4>

</div>

)

}

}

As we can see in the code above, **this**keyword is used in the case of class components.

**Handling state**

Functional components use React hooks to handle state. It uses the useState hook to set the state of a variable inside the component:

Class components: For updating the state, we need to first bind the addStudent function to **this**. Only then, we will be able to use the **setState** function which is used to update the state.

**Summary**:

* Functional components with hooks are concise and more straightforward to code with. They perform exactly as the class component; this implies no difference between the two other than syntax.
* By using just functional components in your project, you drastically eliminate the need to refactor the class component into a functional component when it grows.
* Since classes confuse both people and machines, most especially the this keyword, you don’t have to worry about this anymore in functional components.
* No need for unnecessary method binding like we always do in the class component.
* Sharing stateful logic between components is tedious in a class-based approach.

**Why React keens to interest to use functional components instead class components?**

During compilation, the classes are internally anyway converted to functions. So, if we write directly functional components, this will make algorithm run faster.

**What are the life Cycles of ReactJS?**

1. Initialization
2. State/Property Updates
3. Destruction

**State and Props**:

A React component is a reusable piece of code **receiving** props to communicate (interface) with the outside world, may have states for the internal logic of the component, and **producing** a **UI element**.

**Lifecycle methods**

Each component in React has a lifecycle which you can monitor and manipulate during its three main phases.

The three phases are: **Mounting**, **Updating**, and **Unmounting**.

**Mounting**

Mounting means putting elements into the DOM.

React has four built-in methods that gets called, in this order, when mounting a component:

* constructor()
* getDerivedStateFromProps()
* render()
* componentDidMount()

The constructor() method is called before anything else, when the component is initiated, and it is the natural place to set up the initial state and other initial values.

The getDerivedStateFromProps() method is called right before rendering the element(s) in the DOM. **getDerivedStateFromProps** method updates the state according to the props passed to it.

The render() method is required, and is the method that actually outputs the HTML to the DOM.

The componentDidMount() method is called after the component is rendered. Good place to keep Ajax calls

**Updating**

The next phase in the lifecycle is when a component is *updated*.

A component is updated whenever there is a change in the component's state or props.

React has five built-in methods that gets called, in this order, when a component is updated:

* getDerivedStateFromProps()
* shouldComponentUpdate()
* render()
* getSnapshotBeforeUpdate()
* componentDidUpdate()

In the shouldComponentUpdate() method you can return a Boolean value that specifies whether React should continue with the rendering or not.

The render() method is of course called when a component gets *updated*, it has to re-render the HTML to the DOM, with the new changes.

In the getSnapshotBeforeUpdate() method you have access to the props and state *before* the update, meaning that even after the update, you can check what the values were *before* the update.

If the getSnapshotBeforeUpdate() method is present, you should also include the componentDidUpdate() method, otherwise you will get an error.

The componentDidUpdate method is called after the component is updated in the DOM.

**Unmounting**

The next phase in the lifecycle is when a component is removed from the DOM, or *unmounting* as React likes to call it.

React has only one built-in method that gets called when a component is unmounted:

componentWillUnmount()

The componentWillUnmount method is called when the component is about to be removed from the DOM.

**Error Boundary:**

[getDerivedStateFromError()](https://reactjs.org/docs/react-component.html#static-getderivedstatefromerror) or [componentDidCatch()](https://reactjs.org/docs/react-component.html#componentdidcatch).

**Equivalent Hooks:**

componentDidMount

Pass an empty array as the second argument to useEffect() to run only the callback on mount only.

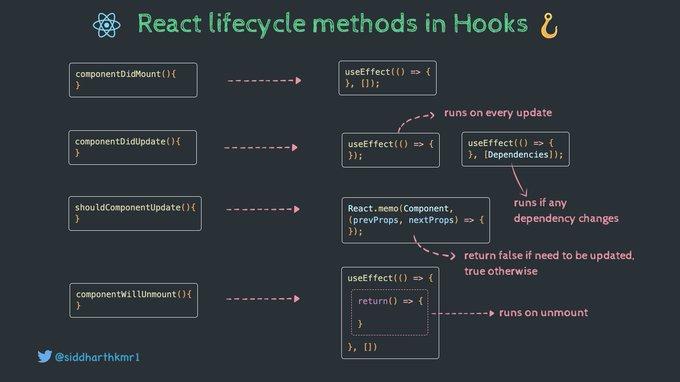
componentDidUpdate

componentDidUpdate() is invoked immediately after updating occurs. This method is not called for the initial render. useEffect runs on every render including the first. So if you want to have a strict equivalent as componentDidUpdate, you have to use useRef to determine if the component has been mounted once. If you want to be even stricter, use useLayoutEffect(), but it fires synchronously. In most cases, useEffect() should be sufficient.

componentWillUnmount

Return a callback in useEffect's callback argument and it will be called before unmounting.

**componentDidMount + componentDidUpdate** is sort of useEffect(callback)



componentDidCatch and getDerivedStateFromError: There are **no** Hook equivalents for these methods yet, but they will be added soon.

**What is the default value of state object in React?**

0

**React hooks:**

useState, useEffect, useContext, useReducer, useMemo, useCallback, useRef, useLayoutEffect

**UseState:**

The React useState Hook allows us to track state in a function component.

State generally refers to data or properties that need to be tracking in an application.

useState accepts an initial state and returns two values:

* The current state.
* A function that updates the state.

setCar(previousState =>

{ return { ...previousState, color: "blue" }

});

**useEffect:**

The useEffect Hook allows you to perform side effects in your components.

Some examples of side effects are: fetching data, directly updating the DOM, and timers.

useEffect accepts two arguments. The second argument is optional.

useEffect(<function>, <dependency>)

**Effect Clean up**

Some effects require clean-up to reduce memory leaks.

Timeouts, subscriptions, event listeners, and other effects that are no longer needed should be disposed.

We do this by including a return function at the end of the useEffect Hook.

1. No dependency passed:

useEffect(() => {

//Runs on every render

});

2. An empty array:

useEffect(() => {

//Runs only on the first render

}, []);

3. Props or state values:

useEffect(() => {

//Runs on the first render

//And any time any dependency value changes

}, [prop, state]);

**React Context/ context API**

* React Context is a way to manage state globally.
* State should be held by the highest parent component in the stack that requires access to the state.
* To illustrate, we have many nested components. The component at the top and bottom of the stack need access to the state.
* To do this without Context, we will need to pass the state as "props" through each nested component. This is called "prop drilling".

const UserContext = createContext()

<UserContext.Provider value={user}>

<h1>{`Hello ${user}!`}</h1>

<Component2 user={user} />

</UserContext.Provider>

In order to use the Context in a child component, we need to access it using the useContext Hook.

const user = useContext(UserContext);

<MyContext.Consumer>

{value => /\* render something based on the context value \*/}

</MyContext.Consumer>

**USEREF:**

* The useRef Hook allows you to persist values between renders.   
  It can be used to store a mutable value that does not cause a re-render when updated.
* It can be used to access a DOM element directly.

const count = useRef(0);

count.current = count.current + 1;

The useRef Hook can also be used to keep track of previous state values.

**When and how to React to use useRef?**

* In plain JavaScript you had to use **getElementById**or **querySelector** to select a DOM node.
* But this is not an ideal solution in React.
* In React you want to use the **useRef**hook or if you’re in a React class component, you want to use **createRef**.

You should avoid using reference calls as much as possible. There are only 3 good reasons why you’d need to use the useRef hook.

* Managing focus, text selection, or media playback
* Triggering imperative animations
* Integrating with third-party DOM libraries

**useLayoutEffect:**

**useLayoutEffect**and **useEffect**are only being executed after a component did mount lifecycle. Both does same.

why should we use this if it’s almost the same thing?

After componentDidMount, first **useLayoutEffect** happens and then **useEffect**

**useLayoutEffect**is identical to **useEffect**, but it’s major key difference is that it gets triggered synchronously after all DOM mutation.

You only want to use this hook when you need to do any DOM changes directly.

This hook is optimized, to allow the engineer to make changes to a DOM node directly before the browser has a chance to paint

**useReducer**:

* The useReducer Hook is similar to the useState Hook.
* It allows for custom state logic for that component.
* The useReducer Hook accepts two arguments.
* useReducer(<reducer>, <initialState>)
* If you find yourself keeping track of multiple pieces of state that rely on complex logic, useReducer may be useful.

The useReducer Hook returns the current state and a dispatch method.

const [todos, dispatch] = useReducer(reducer, initialTodos);

**useCallback:**

* The React useCallback Hook takes function and dependencies, returns a memoized callback function.
* Think of memoization as caching a value so that it does not need to be recalculated.
* This allows us to isolate resource intensive functions so that they will not automatically run on every render.
* The useCallback Hook only runs when one of its dependencies update.
* This can improve performance.

The useCallback and useMemo Hooks are similar. The main difference is that useMemo returns a memoized ***value*** and useCallback returns a memoized ***function***.

**How to solve the React useEffect Hook’s infinite loop patterns?**

UseEffect with no second argument

* If your useEffect function does not contain any dependencies (not even empty array), an infinite loop will occur.
* As this trigger on every update cycle if there are no dependencies.

Using a function as a dependency

* If you pass a method into your useEffect dependency array, React will throw an error, indicating that you have an infinite loop:
* **useCallback** to fix the issue

Using an array as a dependency

Causes for issue:

* Here, recall that React uses shallow comparison to check if the dependency’s reference has changed.
* Since the reference to myArray keeps on changing upon each render, useEffect will trigger the setCount callback
* Therefore, due to myArray's unstable reference value, React will invoke useEffect on every render cycle. Eventually, this causes your application to crash
* **useRef** to fix the issue

Passing an object as a dependency

* Same reasons as bove
* **useMemo** to fix the issue

Passing an incorrect dependency

useEffect(() => {

setCount((count) => count + 1);

}, [count]); //notice that we passed count to this array

[**Is it possible to return empty null undefined in react render function?**](https://stackoverflow.com/questions/42083181/is-it-possible-to-return-empty-in-react-render-function)

Yes. False, null, undefined, and true are valid children. They simply don’t render.

**Dynamically Importing Components with React.lazy**

In JavaScript frameworks like React, it's tempting to statically import a wide range of components. But not all of them are needed immediately; some components only render after a user interacts with the page or once a certain run-time condition has been met.

Some examples of this include:

* Navigable tabs, where each tab renders its own corresponding tab panel.
* Switch statements that cycle through IDs and render the corresponding component.
* Feature-flagged components that render different UI based on some conditions.
* Any rendering logic that's dynamic or conditional in nature.

import { lazy } from 'react';

const MyComponent = lazy(() => import('path/to/component'));

**Suspense** component, which takes a fallback prop and wraps the lazy component: to show spinner or loading…

**Concerns**: Fallback should be lighter

PURE components

Can we update redux store outside react component

Child to parent communication

Difference between ref and state

<https://www.toptal.com/react/interview-questions>

<https://www.ambitionbox.com/profiles/react-developer/interview-questions>

<https://bestofreactjs.com/repo/sudheerj-reactjs-interview-questions-react-miscellaneous>

<https://www.maheshbhusanoor.com/article/top-100-react-redux-interview-question-answers.html>

<https://github.com/sudheerj/reactjs-interview-questions>

<https://dev.to/frontendengineer/react-interview-questions-part-2-mid-senior-level-cal>

<https://www.testgorilla.com/blog/advanced-react-js-interview-questions/>

<https://www.interviewbit.com/react-interview-questions/#how-does-the-performance-of-hooks-differ-in-comparison-with-the-classes>

<https://arc.dev/developer-blog/reactjs-interview-questions/>