**1. DOM**DOM = Document Object Model.   
Document = HTML document.   
Object = Everything inside document i.e. all HTML Tags.  
Model = Layout or Structure.  
So DOM is the layout of the Objects or HTML tags in a document or HTML document.

**2. Virtual DOM**Exact Replica of DOM, upon which React performs all the operations.  
When changes occur in a component's state or props, React creates a new virtual DOM, which is then compared to the previous virtual DOM to identify the changes that need to be made in the real DOM. This process is called reconciliation, and it allows React to reduce the number of DOM manipulations, and improve the performance of the application

**3. JSX**Acronym for JavaScript XML. It is HTML like code written in JavaScript.  
Browsers can't read JSX directly. Browser understands HTML, CSS and JS. JSX is converted into JavaScript using Babel.

**4. Higher Order Component**It is a function that takes a component as an argument and returns a new component. It allows developers to reuse component logic across multiple components.  
Use Case: - HOCs can be used to encapsulate common logic such as fetching data, managing state, or handling events, and then apply that logic to multiple components. This can help to reduce code duplication and make components more modular and reusable.

import React, { useState, useEffect } from 'react';

// This is our Higher-Order Component (HOC) as a custom hook  
const withLoading = (WrappedComponent) => {  
 return function WithLoading(props) {  
 if (props.isLoading) {return <div>Loading...</div>;}  
 return <WrappedComponent {...props} />;  
 };  
};

// Component A without HOC  
function ComponentA() { return <div>Component A Content</div>; }

// Component B without HOC  
function ComponentB() { return <div>Component B Content</div>; }

// Wrapping components with the HOC  
const ComponentWithLoadingA = withLoading(ComponentA);  
const ComponentWithLoadingB = withLoading(ComponentB);

// Usage of wrapped components

function App() {

const [isLoadingA, setIsLoadingA] = useState(true);  
 const [isLoadingB, setIsLoadingB] = useState(false);

useEffect(() => {

// Simulate data fetching delay  
 const timeoutA = setTimeout(() => {  
 setIsLoadingA(false);  
 setIsLoadingB(true);  
 }, 2000);

return () => { clearTimeout(timeoutA); };

}, []);

return (  
 <div>  
 <ComponentWithLoadingA isLoading={isLoadingA} />  
 <ComponentWithLoadingB isLoading={isLoadingB} />  
 </div>  
 );  
}

export default App;

In this example, the HOC is implemented as a custom hook named withLoading, which returns a functional component. The useState and useEffect hooks are used to manage state and side effects in the functional component App. The rest of the code is quite similar to the class component example, but adapted to use functional components and hooks.

Functional components and hooks provide a more concise and readable way to manage state and side effects in React applications.

**5. State**State is an Object that is used to contain data or information about the component. Whenever state changes the component re-renders.

**6. Props**  
Props allow us to pass data from one component to another component.

**7. Children Props**Children props allow us to pass components as data to other components.  
It is used to display whatever you include between the opening and closing tags when invoking a component. For E.g.  
*const Picture = (props) => {  
 return (  
 <div>  
 <img src={props.src}/>  
 {props.children}  
 </div>  
 )  
}*

*return (  
 <div className='container'>  
 <Picture key={picture.id} src={picture.src}>  
 //what is placed here is passed as props.children   
 </Picture>  
 </div>  
 )*

Instead of invoking the component with a self-closing tag <Picture /> if you invoke it will full opening and closing tags <Picture> </Picture> you can then place more code between it.

This de-couples the <Picture> component from its content and makes it more reusable.

**8. React Fragments**Fragments allow us to group a list of children without adding extra nodes to the DOM.  
We can use <React.Fragment></React.Fragment> or its shorter syntax <></>

**9. Lifecycle method of React**  
React contains 3 stages

* Mounting
* Updating
* Un-Mounting

Mounting contains 4 lifecycle methods:

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

Updating contains 5 lifecycle methods:

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

Un-mounting contains 1 lifecycle method

* componentWillUnmount()

**10. Proptype**It allows us to define the expected types of props that are passed to a component. PropTypes validate the props at runtime and help catch bugs and prevent unexpected behavior.

To use PropTypes, you need to import it from the "prop-types" package and define the expected types for each prop in the component. E.g.

*import PropTypes from 'prop-types';  
function Greeting(props) {  
 return <h1>Hello, {props.name}!</h1>;  
}  
Greeting.propTypes = {  
 name: PropTypes.string.isRequired  
};*

In this example, we define a prop type for the "name" prop and specify that it is expected to be a string. We also use the "isRequired" validator to ensure that the prop is passed to the component.  
PropTypes can also be used to validate objects, arrays, and other complex data structures. They support a wide range of validators, including "isRequired", "arrayOf", "objectOf", "shape", and more.

**11. Redux**Redux is a javascript library that is used for state management.

At its core, Redux follows a unidirectional data flow architecture, where the application state is represented by a single object called the "**store**". The store is created by combining multiple "reducers", which are functions that update the state in response to actions.  
Actions in Redux are plain JavaScript objects that describe the changes to be made to the application state. Actions are dispatched to the store, which then invokes the relevant reducers to update the state. The updated state is then passed back to the UI, which can re-render to reflect the changes.

For installation:- *npm install redux react-redux*

*import { createStore } from 'redux';  
const initialState = {count: 0};*

*function reducer(state = initialState, action) {  
 switch (action.type) {  
 case 'INCREMENT': return {count: state.count + 1};  
 case 'DECREMENT': return {count: state.count - 1};  
 default: return state;  
 }  
}  
const store = createStore(reducer);  
export default store;*

**12. Refs**Refs allows us to get a reference to a DOM element. Refs provide a way to access and manipulate the underlying DOM nodes or child components outside of the normal React data flow.  
We can use the useRef() hook to create refs in functional components.

**13. Reconciliation**When a component's props or state change, React compares the new values with the previous values and determines which parts of the UI need to be updated. This process is called Reconcilation.  
React uses a diffing algorithm to compare the previous and current versions of the UI and generate a minimal set of changes that need to be applied. This allows React to avoid unnecessary updates and improve performance.  
During the reconciliation process, React creates a new tree of React elements and compares it with the previous tree. React then determines the differences between the two trees and updates the affected parts of the UI.  
Diffing algorithm follows a heuristic approach with complexity of O(n).  
It is based on 2 assumptions:-

* If an element's type has changed, React assumes that the entire subtree has changed and replaces it with a new subtree.
* The developer can hint at which child elements may be stable across different renders with a "key" prop.

**14. Hooks**  
Hooks are the functions which "hook into" or connect to React state and lifecycle features for function components.

**15. Key prop**  
Keys help React identify which items have changed, are added, or are removed.

**16. useState**   
useState is a Hook that allow us to add React state to function components.

**17. useEffect**It allows us to perform side effects in response to changes in props, state, or other variables.

**18.** **useMemo**It is used to memoize a value, which means that the value is only recomputed when its dependencies change.  
The useMemo() hook takes two arguments: a function that computes the value, and an array of dependencies that the value depends on. The function is only re-run when one of the dependencies changes, and the memoized value is returned from the hook.

*import React, { useMemo } from 'react';  
function MyComponent(props) {  
 const { a, b } = props;  
 const result = useMemo(() => {  
 console.log('Computing result...');  
 return a + b;  
 }, [a, b]);  
return (  
 <div>  
 <p>Result: {result}</p>  
 </div>  
 );  
}*In this example, the useMemo() hook is used to compute the sum of ‘a’ and ‘b’, and the resulting value is stored in the result variable. The function passed to useMemo() is only re-run when either ‘a’ or ‘b’ changes, and the memoized result value is returned from the hook.

**19. useRefs**It is a hook used to create refs.

**20. useCallback**It is a hook that allows you to memoize a function, which means that the function is only recreated when its dependencies change.  
The useCallback() hook takes two arguments: a function to memoize, and an array of dependencies that the function depends on. The memoized function is returned from the hook, and can be passed as a prop or used in other parts of your component.  
*import React, { useCallback } from 'react';*  
 *function MyComponent(props) {  
 const { onClick } = props;  
 const handleClick = useCallback(() => {  
 console.log('Button clicked!');  
 onClick();  
 }, [onClick]);  
return (  
 <div>  
 <button onClick={handleClick}>Click me</button>  
 </div>  
 );  
 }*  
In this example, the useCallback() hook is used to memoize the handleClick() function, which is called when the button is clicked. The function passed to useCallback() is only recreated when the onClick prop changes, and the memoized function is returned from the hook.

**21. Creating Custom Hook***import { useState, useEffect } from 'react';  
function useFetch(url) {  
 const [data, setData] = useState(null);  
 const [error, setError] = useState(null);  
 const [loading, setLoading] = useState(true);  
 useEffect(() => {  
 const fetchData = async () => {  
 try {  
 const response = await fetch(url);  
 const json = await response.json();  
 setData(json);  
 setLoading(false);  
 } catch (error) {  
 setError(error);  
 setLoading(false);  
 }  
 };  
 fetchData();  
 }, [url]);   
 return { data, error, loading };*

*}  
export default useFetch;*

In this example, the useFetch() hook is created to fetch data from an API endpoint and return the data, error, and loading status.  
The useFetch() hook is then exported as a module, and can be used in other components.

*import useFetch from './useFetch';  
function MyComponent() {  
 const { data, error, loading } = useFetch('https://api.example.com/data');  
 if (loading) {return <div>Loading...</div>;}  
 if (error) {return <div>Error: {error.message}</div>; }  
return (  
 <div>  
 {data && JSON.stringify(data)</pre>}  
 </div>  
 );*

*}*

**22. When does React Component re-renders?**In React, a component re-renders when its state or props change.

**23. Context API**In React, the Context API is a feature that allows us to share data between components without passing the data down through props. Context provides a way to avoid "prop drilling", where props are passed through many levels of components to reach a deeply nested component.

**24. Context API v/s Redux**Here are some of the key differences between the Context API and Redux:

* Complexity: The Context API is simpler and easier to use than Redux, making it a good choice for small to medium-sized applications. Redux, on the other hand, has a steeper learning curve and requires more setup, but can provide more advanced features for larger applications.
* Centralized Store: Redux includes a centralized store that holds all of the application's state, while the Context API allows you to create multiple context objects to store state in different parts of the application.
* Middleware: Redux includes middleware, which allows you to intercept and modify actions and add additional functionality to the store. The Context API does not include middleware.
* Debugging: Redux includes time travel debugging, which allows you to step back and forth through the application's state changes, making it easier to debug and understand complex applications.

Overall, the Context API can be a good choice for small to medium-sized applications with a limited amount of shared state, while Redux is better suited for larger and more complex applications with a significant amount of shared state and advanced state management needs. However, the choice between the two ultimately depends on the specific requirements and complexity of the application.

**25. Pure Components**Pure components are a type of component that only re-render when their props or state change, which can help reduce unnecessary re-renders and improve the overall efficiency of a React application. Pure components can be created using the React.PureComponent class or by using the React.memo higher-order component.

**26. Error Boundries**Error Boundries are react components that   
 a. catches javascript errors anywhere in their child component tree,  
 b. log those errors.  
 c. display a fallback UI instead of component tree that crashed

A class component becomes an error boundary if it defines either (or both) of the lifecycle methods static getDerivedStateFromError() or componentDidCatch().  
getDerivedStateFromError() to render a fallback UI after an error has been thrown. componentDidCatch() to log error information.

**27. Portals**Portals provide a way to render children into a DOM node that exists outside the DOM hierarchy of the parent component.  
We can create portal using ReactDOM.createPortal  
e.g. ReactDOM.createPortal(<child-component>,target-dom-node)  
ReactDOM.createPortal takes two arguments: the child component to render, and the target DOM node where the child component should be mounted.

**28. Fetching data from Service**

Get Request

*fetch("url").then(response=>{console.log(response)}).catch(error => {console.log(error)})*

Post Request

*fetch('https://example.com/profile', {  
 method: 'POST', // or 'PUT'  
 headers: { 'Content-Type': 'application/json' },  
 body: JSON.stringify(data),  
}).then(response => { console.log(response)}).catch(error => { console.log(error)})*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Get request

*axios.get(URL).then((response) => {console.log(response.data)});*

Post Request

*axios.post(URL, {  
 title: "Hello World!",  
 body: "This is a new post."  
}).then((response) => {console.log(response.data)});*

Axios is better than fetch because

1. Axios has better error handling. It throws 400 and 500 range errors for us. Unlike the Fetch API, where we have to check the status code and throw the error yourself.
2. With axios we don't need to set headers and converting request body to JSON string. Axios do all these things for us which lacks in Fetch.

**29. Controlled v/s Uncontrolled Component  
Controlled Component** is one that takes its current value through props and notifies changes through callbacks like "onChange"  
In a controlled component, the value of the input element is controlled by React.  
for example:- <input type="text" value={value} onChange={handleChange} />

**Uncontrolled Component** query the DOM using a "ref" to find its current value when we need it.  
In an uncontrolled component, the value of the input element is handled by the DOM itself.  
 for example:- <input type="text" defaultValue="foo" ref={inputRef} />  
// Use `inputRef.current.value` to read the current value of <input>

**30. Strict Mode**  
Strict Mode is used to highlight potential problems in an application. It performs additional checks on the application.  
StrictMode can helps with:

* Identifying components with unsafe lifecycles
* Warning about legacy string ref API usage
* Warning about deprecated findDOMNode usage
* Detecting unexpected side effects
* Detecting legacy context API
* Ensuring reusable state

31. Lazy Loading.

1. Lazy Loading is also called Code Splitting.
2. React has a concept of Component Based Architecture which means that instead of building our application as a whole we can split our application into multiple reuseable components.
3. Now when we do production build at that time webpack(under the hood) will bundle all of the component and it will generate bundle.js file.
4. On initial request, If our application is huge then the JS bundle will take a long time to download & execute the script file which in turn lead to performance loss.
5. To avoid the performance loss we use code splitting. React Lazy & Suspense are preferred way to do code splitting in react applications.
6. React.lazy taskes a function that must call a dynamic import().
7. The lazy component should then be rendered inside the suspense component, which allow us to show fallback content(such as loading indicator) while we wait for lazy component to load.

For Example:-

* Let's assume we have a "Profile" component and we have applied code splitting.
* Whenever we visits "Profile" page at that time React will dynamically load that component.
* It simply means that code for "Profile" component is not initially included in initial "bundle.js" file that gets sent to the client.
* Instead, a seperate chunk file is created for the Profile component which only gets downloaded when user visits that component.

Syntax

Lazy Loading using <Suspense>

1. Import the component that will be loaded lazily. e.g. const UserProfile = React.lazy(() => import('./UserProfile.js'));

2. Wrap the imported component with <Suspense>.   
 <Suspense fallback = {<p>Loading....</p>}> <UserProfile/> </Suspense>.

3. fallback prop accepts a component that will be visible until the actual component loads.

**32. Authentication in React js.**

One of the promising and best way is to use JWT. We will store the JWT token & store it in local storage. This token can be appended in header to every or some axios request in Header.

Case 1: If we want to append the token to every request.

*axios.interceptors.request.use(request=>{  
 request.headers.authorization = localStorage.getItem("jwt\_token");  
 });*

Case 2 : If we want to append token to only some requests

*const authAxios = axios.create({  
 baseUrl: apiUrl,  
 headers: { Authorization: localStorage.getItem("jwt\_token") }  
 })*

**33. Storing & fetching data in local storage & session storage** localStorage.setItem("jwt\_token", token);  
 localStorage.getItem("jwt\_token");  
 localStorage.clear();

sessionStorage.setItem("jwt\_token", token);  
 sessionStorage.getItem("jwt\_token");  
 sessionStorage.clear();

**34. What are Interceptors**Interceptors are the default configurations that are added automatically to every api request or response that a user receives.

**35. Routing in React js**