**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.

**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**