

Q1. What is React.js?

React is a JavaScript library created for building fast and interactive user interfaces for web and mobile applications. It is an open-source, component-based, front-end library responsible only for the application view layer.

The main objective of ReactJS is to develop User Interfaces (UI) that improves the speed of the apps. It uses virtual DOM (JavaScript object), which improves the performance of the app. The JavaScript virtual DOM is faster than the regular DOM. We can use ReactJS on the client and server-side as well as with other frameworks. It uses component and data patterns that improve readability and helps to maintain larger apps.

Q2. How React works?

While building client-side apps, a team at Facebook developers realized that the DOM is slow (The Document Object Model (DOM) is an application programming interface (API) for HTML and XML documents. It defines the logical structure of documents and the way a document is accessed and manipulated). So, to make it faster, React implements a virtual DOM that is basically a DOM tree representation in Javascript. So when it needs to read or write to the DOM, it will use the virtual representation of it. Then the virtual DOM will try to find the most efficient way to update the browsers DOM.

Unlike browser DOM elements, React elements are plain objects and are cheap to create. React DOM takes care of updating the DOM to match the React elements. The reason for this is that JavaScript is very fast and it is worth keeping a DOM tree in it to speedup its manipulation.

Q3. What are Components in React?



Components are the building blocks of any React app and a typical React app will have many of these. Simply put, a component is a JavaScript class or function that optionally accepts inputs i.e. properties(props) and returns a React element that describes how a section of the UI (User Interface) should appear.

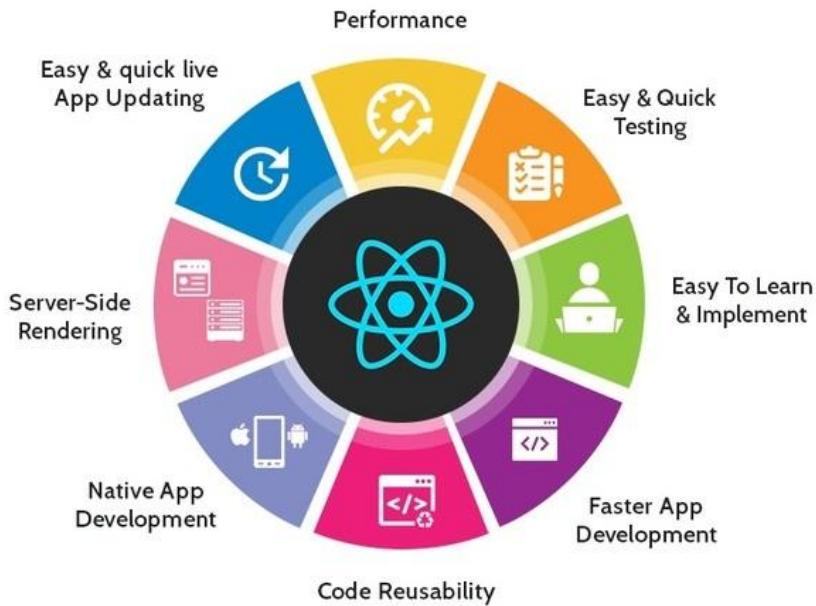
A react application is made of multiple components, each responsible for rendering a small, reusable piece of HTML. Components can be nested within other components to allow complex applications to be built out of simple building blocks. A component may also maintain internal

state – for example, a TabList component may store a variable corresponding to the currently open tab.

Example: Class Component

```
class Welcome extends React.Component {  
  render() {  
    return <h1>Hello, World!</h1>  
  }  
}
```

Q4. List some of the major advantages and limitations of React?



Advantages

- It relies on a virtual-dom to know what is really changing in UI and will re-render only what has really changed, hence better performance wise
- JSX makes components/blocks code readable. It displays how components are plugged or combined with.
- React data binding establishes conditions for creation dynamic applications.
- Prompt rendering. Using comprises methods to minimise number of DOM operations helps to optimise updating process and accelerate it. Testable. React native tools are offered for testing, debugging code.
- SEO-friendly. React presents the first-load experience by server side rendering and connecting event-handlers on the side of the user:
 - React.renderComponentToString is called on the server.
 - React.renderComponent() is called on the client side.
 - React preserves markup rendered on the server side, attaches event handlers.

Limitations

- Learning curve. Being not full-featured framework it is required in-depth knowledge for integration user interface free library into MVC framework.
- View-orientedness is one of the cons of ReactJS. It should be found 'Model' and 'Controller' to resolve 'View' problem.

- Not using isomorphic approach to exploit application leads to search engines indexing problems.

Q5. What is JSX and how JSX can help applications in React.js?

JSX allows us to write HTML elements in JavaScript and place them in the DOM without any `createElement()` or `appendChild()` methods. JSX converts HTML tags into react elements. React uses JSX for templating instead of regular JavaScript. It is not necessary to use it, however, following are some pros that come with it.

- It is faster because it performs optimization while compiling code to JavaScript.
- It is also type-safe and most of the errors can be caught during compilation.
- It makes it easier and faster to write templates.

Example:

```
import React from 'react'

class App extends React.Component {

  render() {
    return (
      <div>
        Hello World!
      </div>
    )
  }
}

export default App
```

JSX is a JavaScript Expression

JSX expressions are JavaScript expressions too. When compiled, they actually become regular JavaScript objects. For instance, the code below:

```
const hello = <h1 className = "greet"> Hello World </h1>
```

will be compiled to

```
const hello = React.createElement {
  type: "h1",
  props: {
    className: "greet",
    children: "Hello World"
  }
}
```

Since they are compiled to objects, JSX can be used wherever a regular JavaScript expression can be used.

Q6. What is ReactDOM?

`ReactDOM()` is a package that provides DOM specific methods that can be used at the top level of a web app to enable an efficient way of managing DOM elements of the web page.

`ReactDOM` provides the developers with an API containing following methods and a few more.

- `render()`
- `findDOMNode()`
- `unmountComponentAtNode()`
- `hydrate()`
- `createPortal()`

Before **v0.14** React Dom was part of React. The reason React and ReactDOM were split into two libraries was due to the arrival of React Native. React contains functionality utilised in web and mobile apps. ReactDOM functionality is utilised only in web apps.

ReactDOM uses observables thus provides an efficient way of DOM handling. ReactDOM can be used in both client-side and server-side.

Example:

```
// index.js

import React from 'react'
import ReactDOM from 'react-dom'
import App from './App/App'

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
)
```

To use the ReactDOM in any React web app we must first import ReactDOM from the react-dom package by using the following code snippet:

```
import ReactDOM from 'react-dom'
```

a.) ReactDOM.render() Function

This function is used to render a single React Component or several Components wrapped together in a Component or a div element. This function uses the efficient methods of React for updating the DOM by being able to change only a subtree, efficient diff methods etc. This function returns a reference to the component or null if a stateless component was rendered.

ReactDOM.render() replaces the child of the given container if any. It uses highly efficient diff algorithm and can modify any subtree of the DOM.

```
ReactDOM.render(element, container, callback)
```

- **element:** This parameter expects a JSX expression or a React Element to be rendered.
- **container:** This parameter expects the container in which the element has to be rendered.
- **callback:** This is an optional parameter that expects a function that is to be executed once the render is complete.

b.) findDOMNode() Function

This function is generally used to get the DOM node where a particular React component was rendered. This method is very less used as the following can be done adding a ref attribute to each component itself.

findDOMNode() function can only be implemented upon mounted components thus Functional components can not be used in findDOMNode() method.

```
ReactDOM.findDOMNode(component)
```

This method takes a single parameter component which expects a React Component to be searched in the Browser DOM. This function returns the DOM node where the component was rendered on success otherwise null.

c.) unmountComponentAtNode() Function

This function is used to unmount or remove the React Component that was rendered to a particular container.

```
ReactDOM.unmountComponentAtNode(container)
```

This method takes a single parameter container which expects the DOM container from which the React component has to be removed. This function returns true on success otherwise false.

d.) **hydrate()** Function

This method is equivalent to the render() method but is implemented while using server-side rendering.

```
ReactDOM.hydrate(element, container, callback)
```

- **element**: This parameter expects a JSX expression or a React Component to be rendered.
- **container**: This parameter expects the container in which the element has to be rendered.
- **callback**: This is an optional parameter that expects a function that is to be executed once the render is complete.

This function attempts to attach event listeners to the existing markup and returns a reference to the component or null if a stateless component was rendered.

e.) **createPortal()** Function

Usually, when an element is returned from a component's render method, it's mounted on the DOM as a child of the nearest parent node which in some cases may not be desired. Portals allow us to render a component into a DOM node that resides outside the current DOM hierarchy of the parent component.

```
ReactDOM.createPortal(child, container)
```

- **child**: This parameter expects a JSX expression or a React Component to be rendered.
- **container**: This parameter expects the container in which the element has to be rendered.

Q7. What is the difference between ReactDOM and React?

```
import React from 'react' /* importing react */
import ReactDOM from 'react-dom' /* importing react-dom */
```

```
class MyComponent extends React.Component {

  render() {
    return <div>Hello World</div>
  }
}
```

```
ReactDOM.render(<MyComponent />, node)
```

React package contains: React.createElement(), React.createClass(),

React.Component(), React.PropTypes(), React.Children()

ReactDOM package contains: ReactDOM.render(), ReactDOM.unmountComponentAtNode(),

ReactDOM.findDOMNode(), and react-dom/server that including:

ReactDOMServer.renderToString() and ReactDOMServer.renderToStaticMarkup().

The ReactDOM module exposes DOM-specific methods, while React has the core tools intended to be shared by React on different platforms (e.g. React Native).

Q8. What are the differences between a class component and functional component?

Functional Components

- Functional components are basic JavaScript functions. These are typically arrow functions but can also be created with the regular function keyword.
- Sometimes referred to as `stateless` components as they simply accept data and display them in some form; that is they are mainly responsible for rendering UI.

- React lifecycle methods (for example, `componentDidMount()`) cannot be used in functional components.
- There is no render method used in functional components.
- These are mainly responsible for UI and are typically presentational only (For example, a Button component).
- Functional components can accept and use props.
- Functional components should be favored if you do not need to make use of React state.

Example:

```
const ClockUsingHooks = props => {
  const [time, setTime] = useState(new Date())

  const changeTime = () => {
    setTime(new Date())
  }

  useEffect(() => {
    const tick = setInterval(() => {
      changeTime()
    }, 1000)
    return () => clearInterval(tick)
  })
  return (
    <div className="clock">
      <h1>Hello! This is a function component clock.</h1>
      <h2>It is {time.toLocaleTimeString()}.</h2>
    </div>
  )
}

export default ClockUsingHooks
```

Class Components

- Class components make use of ES6 class and extend the `Component` class in React.
- Sometimes called `stateful` components as they tend to implement logic and state.
- React lifecycle methods can be used inside class components (for example, `componentDidMount()`).
- We pass `props` down to class components and access them with `this.props`.
- Class-based components can have `refs` to underlying DOM nodes.
- Class-based components can use `shouldComponentUpdate()` and `PureComponent()` performance optimisation techniques.

Example:

```
class ClockUsingClass extends React.Component {
  constructor(props) {
    super(props)
    this.state = { date: new Date() }
  }

  componentDidMount() {
    this.time = setInterval(() => {
      this.changeTime()
    }, 1000)
  }
}
```

```

componentWillUnmount() {
    clearInterval(this.time)
}

changeTime() {
    this.setState({ date: new Date() })
}

render() {
    return (
        <div className="clock">
            <h1>Hello! This is a class component clock.</h1>
            <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
        </div>
    )
}
}

export default ClockUsingClass

```

Q9. What is the difference between state and props?

State

This is data maintained inside a component. It is local or owned by that specific component. The component itself will update the state using the `setState()` function.

Example:

```

class Employee extends React.Component {
    constructor() {
        this.state = {
            id: 1,
            name: "Alex"
        }
    }

    render() {
        return (
            <div>
                <p>{this.state.id}</p>
                <p>{this.state.name}</p>
            </div>
        )
    }
}

export default Employee

```

Props

Data passed in from a parent component. `props` are read-only in the child component that receives them. However, callback functions can also be passed, which can be executed inside the child to initiate an update.

Example:

```

class ParentComponent extends Component {
    render() {
        return (

```

```

        <ChildComponent name="First Child" />
    )
}

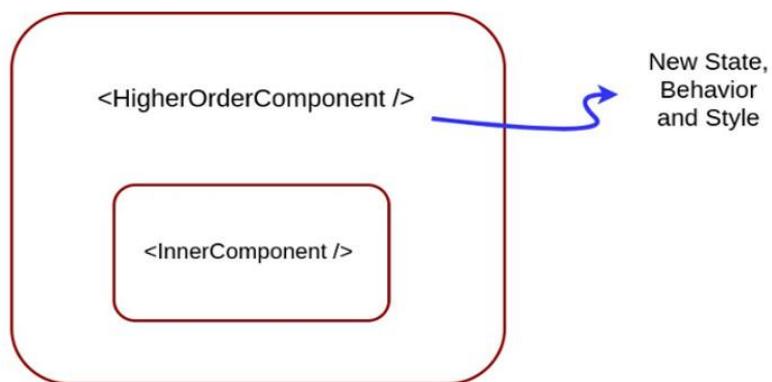
const ChildComponent = (props) => {
  return <p>{props.name}</p>
}

```

Difference between State and Props

| Props | State |
|---|--|
| Props are read-only. | State changes can be asynchronous. |
| Props are immutable. | State is mutable. |
| Props allow you to pass data from one component to other components as an argument. | State holds information about the components. |
| Props can be accessed by the child component. | State cannot be accessed by child components. |
| Props are used to communicate between components. | States can be used for rendering dynamic changes with the component. |
| Stateless component can have Props. | Stateless components cannot have State. |
| Props make components reusable. | State cannot make components reusable. |
| Props are external and controlled by whatever renders the component. | The State is internal and controlled by the React Component itself. |

Q10. How would you create Higher Order Components (HOCs) in React.js?



A higher-order component is a function that takes a component and returns a new component. A higher-order component (HOC) is the advanced technique in React.js for reusing a component logic. Higher-Order Components are not part of the React API. They are the pattern that emerges from React's compositional nature. The component transforms props into UI, and a higher-order component converts a component into another component. The examples of HOCs are Redux's connect and Relay's createContainer.

```

// HOC.js

import React, {Component} from 'react'

export default function Hoc(HocComponent) {
    return class extends Component{
        render() {
            return (
                <div>
                    <HocComponent></HocComponent>
                </div>
            )
        }
    }
}

// App.js

import React, { Component } from 'react'
import Hoc from './HOC'

class App extends Component {

    render() {
        return (
            <div>
                Higher-Order Component Example!
            </div>
        )
    }
}

App = Hoc(App)
export default App

```

Notes

- We do not modify or mutate components. We create new ones.
- A HOC is used to compose components for code reuse.
- A HOC is a pure function. It has no side effects, returning only a new component.

Q11. What is PureComponent?

Pure Components in React are the components which do not re-renders when the value of state and props has been updated with the same values. If the value of the previous state or props and the new state or props is the same, the component is not re-rendered. Pure Components restricts the re-rendering ensuring the higher performance of the Component

Features of React Pure Components

- Prevents re-rendering of Component if props or state is the same
- Takes care of `shouldComponentUpdate()` implicitly
- State() and Props are Shallow Compared
- Pure Components are more performant in certain cases

Similar to Pure Functions in JavaScript, a React component is considered a Pure Component if it renders the same output for the same state and props value. React provides the `PureComponent`

base class for these class components. Class components that extend the `React.PureComponent` class are treated as pure components.

It is the same as Component except that Pure Components take care of `shouldComponentUpdate()` by itself, it does the *shallow comparison* on the state and props data. If the previous state and props data is the same as the next props or state, the component is not Re-rendered.

React Components re-renders in the following scenarios:

1. `setState()` is called in Component
2. `props` values are updated
3. `this.forceUpdate()` is called

In the case of Pure Components, the React components do not re-render blindly without considering the updated values of React `props` and `state`. If updated values are the same as previous values, render is not triggered.

Stateless Component

```
import { pure } from 'recompose'

export default pure ( (props) => {
  return 'Stateless Component Example'
})
```

Stateful Component

```
import React, { PureComponent } from 'react'

export default class Test extends PureComponent{
  render() {
    return 'Stateful Component Example'
  }
}
```

Example:

```
class Test extends React.PureComponent {
  constructor(props) {
    super(props)
    this.state = {
      taskList: [
        { title: 'Excercise' },
        { title: 'Cooking' },
        { title: 'Reacting' },
      ]
    }
  }
  componentDidMount() {
    setInterval(() => {
      this.setState((oldState) => {
        return { taskList: [...oldState.taskList] }
      })
    }, 1000)
  }
  render() {
    console.log("TaskList render() called")
    return (<div>
      {this.state.taskList.map((task, i) => {
        return (<Task
```

```

        key={i}
        title={task.title}
      />
    }) }
  </div>
}
}

class Task extends React.Component {
  render() {
    console.log("task added")
    return (<div>
      {this.props.title}
    </div>
  )
}
}

ReactDOM.render(<Test />, document.getElementById('app'))

```

Q12. Why to use *PureComponent*? When to use *PureComponent* over Component?

Both functional-based and class-based components have the same downside: they always re-render when their parent component re-renders even if the props do not change.

Also, class-based components always re-render when its state is updated (`this.setState()` is called) even if the new state is equal to the old state. Moreover, when a parent component re-renders, all of its children are also re-rendered, and their children too, and so on.

That behaviour may mean a lot of wasted re-renderings. Indeed, if our component only depends on its props and state, then it shouldn't re-render if neither of them changed, no matter what happened to its parent component.

That is precisely what *PureComponent* does - it stops the vicious re-rendering cycle.

PureComponent does not re-render unless its props and state change.

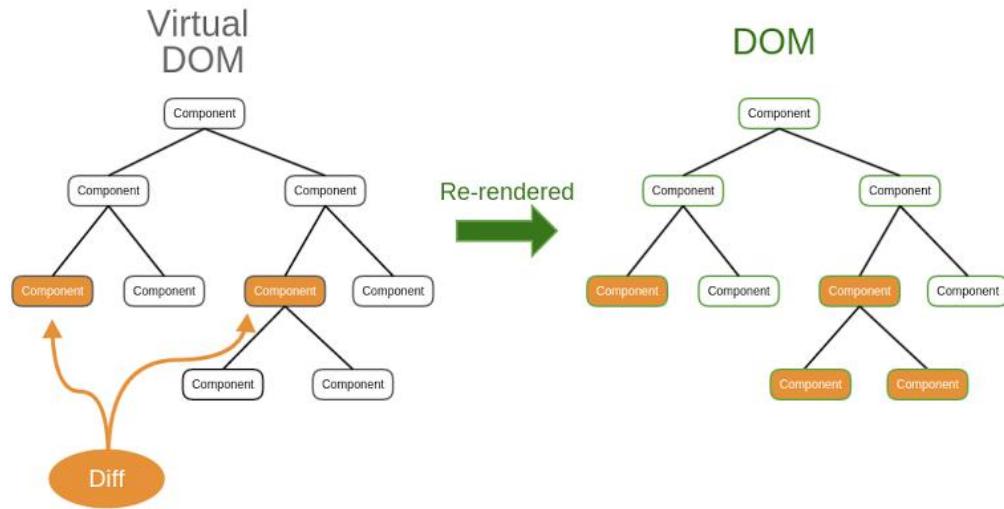
When to use *PureComponent*

- We want to avoid re-rendering cycles of component when its props and state are not changed, and
- The state and props of component are immutable, and
- We do not plan to implement own `shouldComponentUpdate()` lifecycle method.

On the other hand, we should not use `PureComponent()` as a base component if:

- props or state are not immutable, or
- Plan to implement own `shouldComponentUpdate()` lifecycle method.

Q13. How Virtual-DOM is more efficient than Dirty checking?



React Virtual DOM

In React, Each time the DOM updates or data of page changes, a new Virtual DOM representation of the user interface is made. It is just a lightweight copy or DOM.

Virtual DOM in React has almost same properties like a real DOM, but it can not directly change the content on the page. Working with Virtual DOM is faster as it does not update anything on the screen at the same time. In a simple way, Working with Virtual DOM is like working with a copy of real DOM nothing more than that.

Updating virtual DOM in ReactJS is faster because ReactJS uses

1. It is efficient diff algorithm.
2. It batched update operations
3. It efficient update of sub tree only
4. It uses observable instead of dirty checking to detect change

How Virtual DOM works in React

When we render a JSX element, each virtual DOM updates. This approach updates everything very quickly. Once the Virtual DOM updates, React matches the virtual DOM with a virtual DOM copy that was taken just before the update. By Matching the new virtual DOM with pre-updated version, React calculates exactly which virtual DOM has changed. This entire process is called **diffing**.

When React knows which virtual DOM has changed, then React updated those objects. and only those object, in the real DOM. React only updates the necessary parts of the DOM. React's reputation for performance comes largely from this innovation.

In brief, here is what happens when we update the DOM in React:

1. The entire virtual DOM gets updated.
2. The virtual DOM gets compared to what it looked like before you updated it. React matches out which objects have changed.
3. The changed objects and the changed objects only get updated on the real DOM.
4. Changes on the real DOM cause the screen to change finally.

Q14. Why is `setState()` in React *async* instead of *sync*?

Even if state is updated synchronously, props are not, it means we do not know props until it re-renders the parent component. The objects provided by React (`state`, `props`, `refs`) are consistent with each other and if you introduce a synchronous `setState` you could introduce some bugs.

`setState()` does not immediately mutate `this.state()` but creates a pending state transition. Accessing `this.state()` after calling this method can potentially return the existing value. There is no guarantee of synchronous operation of calls to `setState()` and calls may be batched for performance gains.

This is because `setState()` alters the state and causes rerendering. This can be an expensive operation and making it synchronous might leave the browser unresponsive. Thus the `setState()` calls are asynchronous as well as batched for better UI experience and performance.

Q15. What are controlled and uncontrolled components in React?

In a controlled component, form data is handled by a React component. The alternative is uncontrolled components, where form data is handled by the DOM itself.

Controlled Components

In a controlled component, the form data is handled by the state within the component. The state within the component serves as “the single source of truth” for the input elements that are rendered by the component.

Example:

```
import React, { Component } from 'react'

class App extends Component {
  state = {
    message: ''
  }
  updateMessage = (newText) => {
    console.log(newText)
    this.setState(() => ({
      message: newText
    }))
  }
  render() {
    return (
      <div className="App">
        <div className="container">
          <input type="text"
            placeholder="Your message here.."
            value={this.state.message}
            onChange={(event) =>
              this.updateMessage(event.target.value)}
            />
          <p>the message is: {this.state.message}</p>
        </div>
      </div>
    )
  }
}

export default App
```

Uncontrolled Components

Uncontrolled components act more like traditional HTML form elements. The data for each input element is stored in the DOM, not in the component. Instead of writing an event handler for all of your state updates, It uses `ref` to retrieve values from the DOM. `Refs` provide a way to access DOM nodes or React elements created in the render method.

```
import React, { Component } from 'react'

class App extends Component {

  constructor(props) {
    super(props)
    this.handleChange = this.handleChange.bind(this)
    this.input = React.createRef()
  }
  handleChange = (newText) => {
    console.log(newText)
  }
  render() {
    return (
      <div className="App">
        <div className="container">
          <input type="text"
            placeholder="Your message here.."
            ref={this.input}
            onChange={(event) =>
              this.handleChange(event.target.value)}
          />
        </div>
      </div>
    )
  }
}

export default App
```

Q16. What is `React.cloneElement`?

The `React.cloneElement()` function returns a copy of a specified element. Additional props and children can be passed on in the function. This function is used when a parent component wants to add or modify the prop(s) of its children.

```
React.cloneElement(element, [props], [...children])
```

The `react.cloneElement()` method accepts three arguments.

- `element`: Element we want to clone.
- `props`: props we need to pass to the cloned element.
- `children`: we can also pass children to the cloned element (passing new children replaces the old children).

Example

```
import React from 'react'

export default class App extends React.Component {

  // rendering the parent and child component
  render() {
    return (
      <div>
```

```

<ParentComp>
  <MyButton/>
  <br></br>
  <MyButton/>
</ParentComp>
)
}
}
// The parent component
class ParentComp extends React.Component {
  render() {
    // The new prop to be added.
    let newProp = 'red';
    // Looping over the parent's entire children,
    // cloning each child, adding a new prop.
    return (
      <div>
        {React.Children.map(this.props.children,
          child => {
            return React.cloneElement(child,
              {newProp}, null)
          })
      </div>
    )
  }
}
// The child component
class MyButton extends React.Component {
  render() {
    return <button style =
      {{ color: this.props.newProp }}>
      Hello World!</button>
  }
}

```

Q17. When we should use `React.cloneElement` vs `this.props.children`?

The `React.cloneElement()` works if child is a single React element.

For almost everything `{this.props.children}` is used. Cloning is useful in some more advanced scenarios, where a parent sends in an element and the child component needs to change some props on that element or add things like `ref` for accessing the actual DOM element.

React.Children

Since `{this.props.children}` can have one element, multiple elements, or none at all, its value is respectively a single child node, an array of child nodes or undefined. Sometimes, we want to transform our children before rendering them — for example, to add additional props to every child. If we wanted to do that, we'd have to take the possible types of `this.props.children` into account. For example, if there is only one child, we can not map it.

Example:

```
class Example extends React.Component {
```

```

    render() {
      return <div>
        <div>Children ({this.props.children.length}):</div>
        {this.props.children}
      </div>
    }
}

class Widget extends React.Component {

  render() {
    return <div>
      <div>First <code>Example</code>:</div>
      <Example>
        <div>1</div>
        <div>2</div>
        <div>3</div>
      </Example>
      <div>Second <code>Example</code> with different children:</div>
      <Example>
        <div>A</div>
        <div>B</div>
      </Example>
    </div>
  }
}

```

Output

First Example:

Children (3):

1
2
3

Second Example with different children:

Children (2):

A
B

`children` is a special property of React components which contains any child elements defined within the component, e.g. the `<div>` inside `Example` above. `{this.props.children}` includes those children in the rendered result.

Q18. What is the second argument that can optionally be passed to `setState` and what is its purpose?

A callback function which will be invoked when `setState()` has finished and the component is re-rendered.

The `setState()` is asynchronous, which is why it takes in a second callback function. Typically it's best to use another lifecycle method rather than relying on this callback function, but it is good to know it exists.

```

this.setState(
  { username: 'Alex' },
  () => console.log('setState has finished and the component has re-
rendered.')
)
```

)

The `setState()` will always lead to a re-render unless `shouldComponentUpdate()` returns false. To avoid unnecessary renders, calling `setState()` only when the new state differs from the previous state makes sense and can avoid calling `setState()` in an infinite loop within certain lifecycle methods like `componentDidUpdate()`.

Q19. What is `useState()` in React?

The `useState()` is a Hook that allows to have state variables in functional components.

```
import React, { useState } from 'react'
```

```
const App = () => {
  const [count, setCount] = React.useState(0)

  const handleIncrease = () => {
    setCount(count + 1)
  }

  const handleDecrease = () => {
    setCount(count - 1)
  }

  return (
    <div>
      Count: {count}
      <hr />
      <div>
        <button type="button" onClick={handleIncrease}>
          Increase
        </button>
        <button type="button" onClick={handleDecrease}>
          Decrease
        </button>
      </div>
    </div>
  )
}
```

The `useState()` function takes as argument a value for the initial state. In this case, the count starts out with 0. In addition, the hook returns an array of two values: **count** and **setCount**. It's up to you to name the two values, because they are destructured from the returned array where renaming is allowed.

Q20. What is `useReducer()` in React?

It accepts a reducer function with the application initial state, returns the current application state, then dispatches a function.

Although `useState()` is a Basic Hook and `useReducer()` is an Additional Hook, `useState()` is actually implemented with `useReducer()`. This means `useReducer()` is primitive and we can use `useReducer()` for everything can do with `useState()`. Reducer is so powerful that it can apply for various use cases.

Example:

```

import React, { useReducer } from 'react'

const initialState = 0
const reducer = (state, action) => {
  switch (action) {
    case 'increment': return state + 1
    case 'decrement': return state - 1
    case 'reset': return 0
    default: throw new Error('Unexpected action')
  }
}

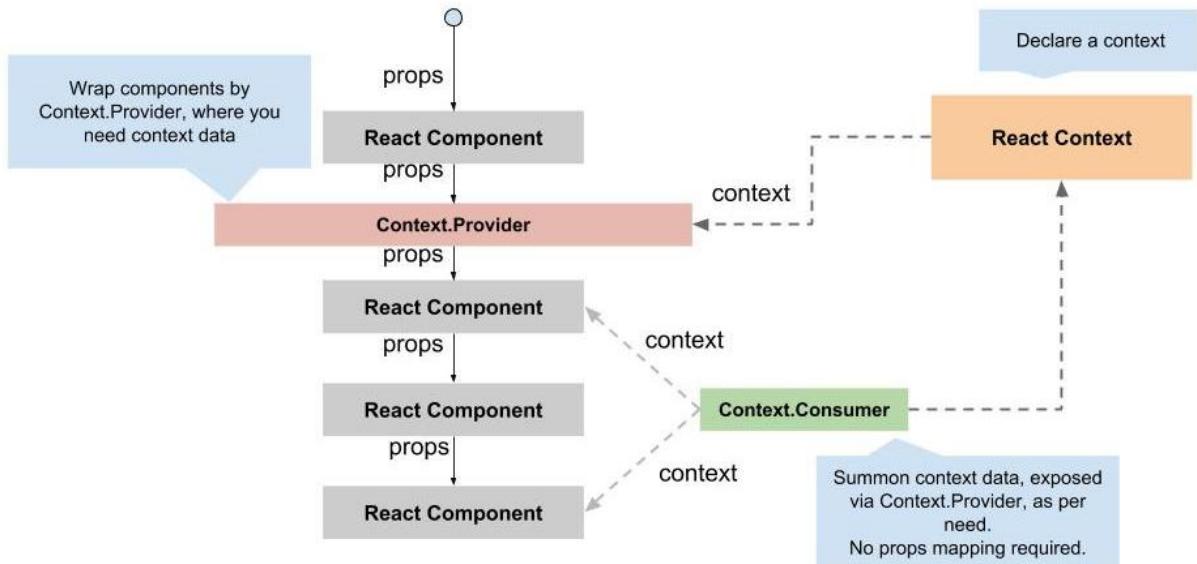
const ReducerExample = () => {
  const [count, dispatch] = useReducer(reducer, initialState)
  return (
    <div>
      {count}
      <button onClick={() => dispatch('increment')}>+1</button>
      <button onClick={() => dispatch('decrement')}>-1</button>
      <button onClick={() => dispatch('reset')}>reset</button>
    </div>
  )
}

export default ReducerExample

```

Here, we first define an initialState and a reducer. When a user clicks a button, it will dispatch an action which updates the count and the updated count will be displayed. We could define as many actions as possible in the reducer, but the limitation of this pattern is that actions are finite.

Q21. What is `useContext()` in React?



The React Context API allows to easily access data at different levels of the component tree, without having to pass data down through `props`.

```
import React from "react"
```

```

import ReactDOM from "react-dom"

// Create a Context
const NumberContext = React.createContext()
// It returns an object with 2 values:
// { Provider, Consumer }

function App() {
  // Use the Provider to make a value available to all
  // children and grandchildren
  return (
    <NumberContext.Provider value={10}>
      <div>
        <Display />
      </div>
    </NumberContext.Provider>
  )
}

function Display() {
  const value = useContext(NumberContext)
  return <div>The answer is {value}.</div>
}

```

Q22. What is difference between `useEffect()` vs `componentDidMount()`?

In react when we use class based components we get access to lifecycle methods(like `componentDidMount`, `componentDidUpdate`, etc). But when we want use a functional component and also we want to use lifecycle methods, then using `useEffect()` we can implement those lifecycle methods.

The `componentDidMount()` and `useEffect()` run after the mount. However `useEffect()` runs after the paint has been committed to the screen as opposed to before. This means we would get a flicker if needed to read from the DOM, then synchronously set state to make new UI.

The `useLayoutEffect()` was designed to have the same timing as `componentDidMount()`. So `useLayoutEffect(fn, [])` is a much closer match to `componentDidMount()` than `useEffect(fn, [])` -- at least from a timing standpoint.

```

//Using a class based component.
import React, { Component } from 'react'

export default class SampleComponent extends Component {
  componentDidMount() {
    // code to run on component mount
  }
  render() {
    return (<div>foo</div>)
  }
}

//Using a functional component
import React, { useEffect } from 'react'

```

```

const SampleComponent = () => {
  useEffect(() => {
    // code to run on component mount
  }, [])
  return (<div>foo</div>)
}
export SampleComponent

```

useEffect() Limitations

When useEffect() is used to get data from server.

- The first argument is a callback that will be fired after browser layout and paint.
Therefore it does not block the painting process of the browser.
- The second argument is an array of values (usually props).
- If any of the value in the array changes, the callback will be fired after every render.
- When it is not present, the callback will always be fired after every render.
- When it is an empty list, the callback will only be fired once, similar to componentDidMount.

Q23. What do you understand by refs in React?

Refs provide a way to access DOM nodes or React elements created in the render method. React Refs are a useful feature that act as a means to reference a DOM element or a class component from within a parent component.

Refs also provide some flexibility for referencing elements within a child component from a parent component, in the form of ref forwarding.

Example:

```

class App extends React.Component {
  constructor(props) {
    super(props)
    // create a ref to store the TextInput DOM element
    this.textInput = React.createRef()
    this.state = {
      value: ''
    }
  }

  // Set the state for the ref
  handleSubmit = e => {
    e.preventDefault()
    this.setState({ value: this.textInput.current.value })
  }

  render() {
    return (
      <div>
        <h1>React Ref - createRef</h1>
        {/** This is what will update **/}
        <h3>Value: {this.state.value}</h3>
        <form onSubmit={this.handleSubmit}>
          {/** Call the ref on <input> so we can use it to update the <h3>
value **/}
          <input type="text" ref={this.textInput} />
          <button>Submit</button>
        </form>
      </div>
    )
  }
}

```

```

        </form>
    </div>
)
}
}

```

When to Use Refs

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

When not to use refs

- Should not be used with functional components because they don't have instances.
- Not to be used on things that can be done declaratively.

Q24. What will happen if you use `setState()` in constructor?

When we use `setState()`, then apart from assigning to the object state react also rerenders the component and all its children. Which we don't need in the constructor, since the component hasn't been rendered anyway.

Inside constructor uses `this.state = {}` directly, other places use `this.setState({ })`

Example:

```

import React, { Component } from 'react'

class Food extends Component {

  constructor(props) {
    super(props)

    this.state = {
      fruits: ['apple', 'orange'],
      count: 0
    }
  }

  render() {
    return (
      <div className = "container">
        <h2> Hello!!!</h2>
        <p> I have {this.state.count} fruit(s)</p>
      </div>
    )
  }
}

```

Q25. What is the difference between DOM and virtual DOM?

DOM

DOM stands for "Document Object Model". The HTML DOM provides an interface (API) to traverse and modify the nodes. It contains methods like `getElementById()` or `removeChild()`. The DOM is represented as a tree data structure. Because of that, the changes and updates to the DOM are fast. But after the change, the updated element and its children have to be re-rendered to update the application UI. The re-rendering or re-painting of the UI is what makes it slow.

Virtual DOM

The virtual DOM is only a virtual representation of the DOM. Everytime the state of our application changes, the virtual DOM gets updated instead of the real DOM.

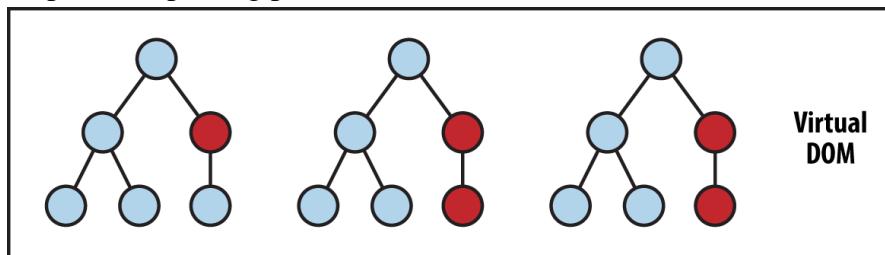
The Virtual DOM is an abstraction of the HTML DOM. It is lightweight and detached from the browser-specific implementation details. Since the DOM itself was already an abstraction, the virtual DOM is, in fact, an abstraction of an abstraction.

Why Virtual DOM is faster

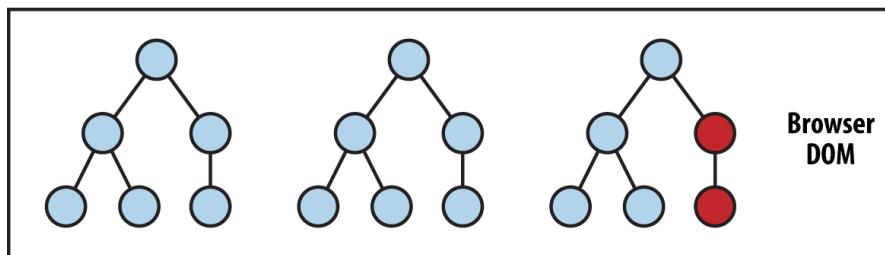
When new elements are added to the UI, a virtual DOM, which is represented as a tree is created. Each element is a node on this tree. If the state of any of these elements changes, a new virtual DOM tree is created. This tree is then compared or “diffed” with the previous virtual DOM tree. Once this is done, the virtual DOM calculates the best possible method to make these changes to the real DOM. This ensures that there are minimal operations on the real DOM. Hence, reducing the performance cost of updating the real DOM.

Pros of Virtual DOM

- Updates process is optimized and accelerated.
- JSX makes components/blocks code readable.
- React data binding establishes conditions for creation dynamic applications.
- Virtual DOM is ideal for mobile first applications.
- Prompt rendering. Using comprises methods to minimize number of DOM operations helps to optimize updating process and accelerate it.



State Change → Compute Diff → Re-render



Q26. When should we use arrow functions with React?

Arrows prevent `this` bugs

Arrow functions do not redefine the value of `this` within their function body. This makes it a lot easier to predict their behavior when passed as callbacks, and prevents bugs caused by use of `this` within callbacks. Using inline arrow functions in function components is a good way to achieve some decoupling.

Example:

```
import React from 'react'
import ReactDOM from 'react-dom'

class Button extends React.Component {
```

```

render() {
  return (
    <button onClick={this.handleClick} style={this.state}>
      Set background to red
    </button>
  )
}

handleClick = () => {
  this.setState({ backgroundColor: 'red' })
}
}

ReactDOM.render(
  <Button />,
  document.getElementById('root')
)

```

1. When we use `this` it generates a new function on every render, which will obviously have a new reference.
2. If the component we pass this generated function to is extending `PureComponent()`, it will not be able to bail out on rerendering, even if the actual data has not changed.

Q27. Differentiate between stateful and stateless components?

Stateful and stateless components have many different names. They are also known as:

- Container vs Presentational components
- Smart vs Dumb components

The literal difference is that one has state, and the other does not. That means the stateful components are keeping track of changing data, while stateless components print out what is given to them via props, or they always render the same thing.

Example: Stateful/Container/Smart component

```

class Main extends Component {
  constructor() {
    super()
    this.state = {
      books: []
    }
  }
  render() {
    <BooksList books={this.state.books} />
  }
}

```

Example: Stateless/Presentational/Dumb component

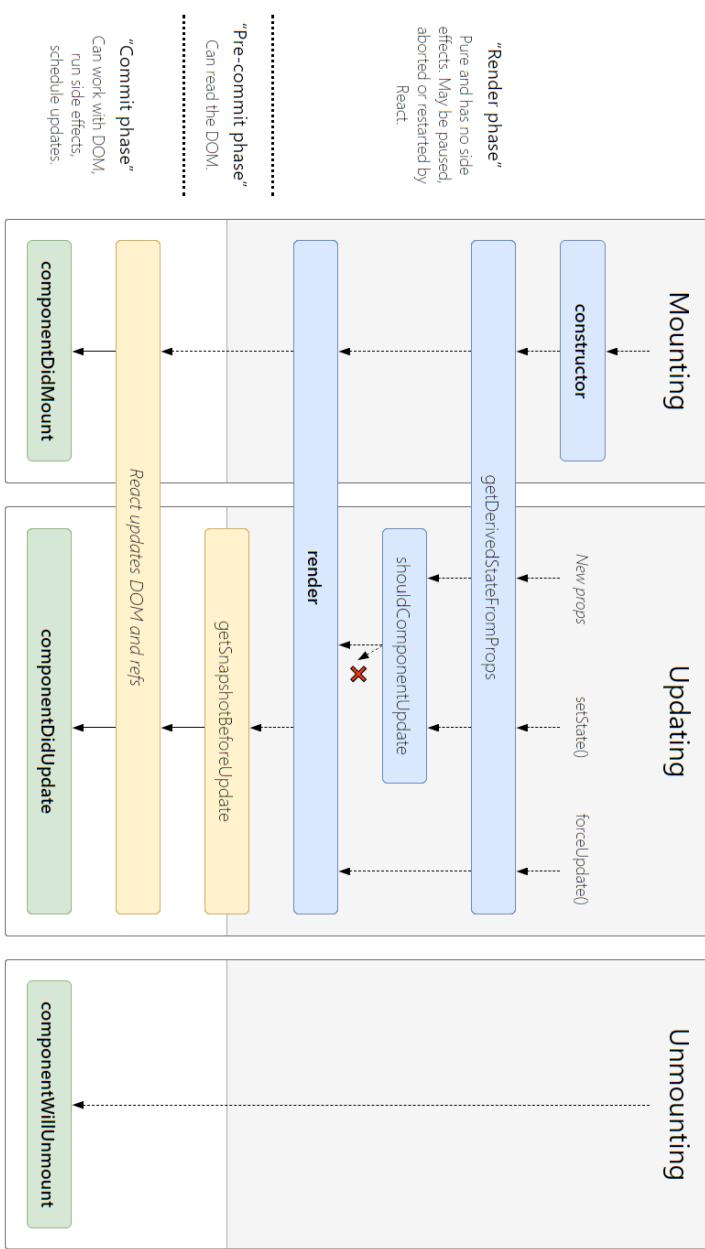
```

const BooksList = ({books}) => {
  return (
    <ul>
      {books.map(book => {
        return <li>book</li>
      })}
    </ul>
  )
}

```

Functional Component or Stateless component

- Functional component is like pure function in JavaScript.
- Functional component is also called as a stateless component.
- The functional component only receives props from parent component and return you JSX elements.



component state.

Class component or statefull component

- React class component is called as a stateful component.
- Stateful component plays with all life cycle methods of React.
- This component will modify state.

When would you use a stateless component

- When you just need to present the props
- When you do not need a state, or any internal variables
- When creating element does not need to be interactive
- When you want reusable code

When would you use a stateful component?

- When building element that accepts user input or element that is interactive on page
- When dependent on state for rendering, such as, fetching data before rendering

- When dependent on any data that cannot be passed down as props

Q28. What are the different phases of React component lifecycle?

React provides several methods that notify us when certain stage of this process occurs. These methods are called the component lifecycle methods and they are invoked in a predictable order. The lifecycle of the component is divided into four phases.

1. Mounting

These methods are called in the following order when an instance of a component is being created and inserted into the DOM:

- constructor()
- getDerivedStateFromProps()
- render()
- componentDidMount()

constructor

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 `constructor()` method is called with the `props`, as arguments, and we should always start by calling the `super(props)` before anything else, this will initiate the parent's constructor method and allows the component to inherit methods from its parent (`React.Component`).

getDerivedStateFromProps

The `getDerivedStateFromProps()` method is called right before rendering the element(s) in the DOM. It takes state as an argument, and returns an object with changes to the state.

Example:

```
class Color extends React.Component {

  constructor(props) {
    super(props)
    this.state = {color: "red"}
  }
  static getDerivedStateFromProps(props, state) {
    return {color: props.favcol}
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.color}</h1>
    )
  }
}
```

```
ReactDOM.render(<Color favcol="yellow"/>, document.getElementById('root'))
```

render()

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

componentDidMount()

The `componentDidMount()` method is called after the component is rendered.

2. 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()`

getDerivedStateFromProps

This is the first method that is called when a component gets updated. This is still the natural place to set the state object based on the initial props.

Example:

```
class Color extends React.Component {  
  
  constructor(props) {  
    super(props)  
    this.state = {color: "red"}  
  }  
  static getDerivedStateFromProps(props, state) {  
    return {color: props.favcol}  
  }  
  changeColor = () => {  
    this.setState({color: "blue"})  
  }  
  render() {  
    return (  
      <div>  
        <h1>My Favorite Color is {this.state.color}</h1>  
        <button type="button" onClick={this.changeColor}>Change color</button>  
      </div>  
    )  
  }  
}  
  
ReactDOM.render(<Color favcol="yellow"/>, document.getElementById('root'))
```

shouldComponentUpdate

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

```
class Color extends React.Component {  
  
  constructor(props) {  
    super(props)  
    this.state = {color: "red"}  
  }  
  shouldComponentUpdate() {  
    return false  
  }  
  changeColor = () => {  
    this.setState({color: "blue"})  
  }  
  render() {  
    return (
```

```

        <div>
          <h1>My Favorite Color is {this.state.color}</h1>
          <button type="button" onClick={this.changeColor}>Change color</button>
        </div>
      )
    }
}

ReactDOM.render(<Color />, document.getElementById('root'))

```

render()

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.

getSnapshotBeforeUpdate

In the `getSnapshotBeforeUpdate()` method we have access to the `props` and `state` before the update, meaning that even after the update, we can check what the values were before the update. If the `getSnapshotBeforeUpdate()` method is present, we should also include the `componentDidUpdate()` method, otherwise it will throw an error.

Example:

```

class Color extends React.Component {

  constructor(props) {
    super(props)
    this.state = {color: "red"}
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({color: "yellow"})
    }, 1000)
  }
  getSnapshotBeforeUpdate(prevProps, prevState) {
    document.getElementById("div1").innerHTML =
      "Before the update, the favorite was " + prevState.color
  }
  componentDidUpdate() {
    document.getElementById("div2").innerHTML =
      "The updated favorite is " + this.state.color
  }
  render() {
    return (
      <div>
        <h1>My Favorite Color is {this.state.color}</h1>
        <div id="div1"></div>
        <div id="div2"></div>
      </div>
    )
  }
}

ReactDOM.render(<Color />, document.getElementById('root'))

```

componentDidUpdate

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

Example:

```

class Color extends React.Component {
  constructor(props) {
    super(props)
    this.state = {color: "red"}
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({color: "yellow"})
    }, 1000)
  }
  componentDidUpdate() {
    document.getElementById("mydiv").innerHTML =
      "The updated favorite is " + this.state.color
  }
  render() {
    return (
      <div>
        <h1>My Favorite Color is {this.state.color}</h1>
        <div id="mydiv"></div>
      </div>
    )
  }
}

```

ReactDOM.render(<Color />, document.getElementById('root'))

3. Unmounting

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

- `componentWillUnmount()`

Example: Click the button to delete the header

```

class Container extends React.Component {
  constructor(props) {
    super(props)
    this.state = {show: true}
  }
  delHeader = () => {
    this.setState({show: false})
  }
  render() {
    let myheader
    if (this.state.show) {
      myheader = <Child />
    }
    return (
      <div>
        {myheader}
        <button type="button" onClick={this.delHeader}>Delete Header</button>
      </div>
    )
  }
}

class Child extends React.Component {
  componentWillMount() {

```

```

        alert("The component named Header is about to be unmounted.")
    }
    render() {
        return (
            <h1>Hello World!</h1>
        )
    }
}

ReactDOM.render(<Container />, document.getElementById('root'))

```

Q29. What is the significance of keys in React?

Keys help React identify which items have changed, are added, or are removed. Keys should be given to the elements inside the array to give the elements a stable identity.

```

function NumberList(props) {

    const numbers = props.numbers
    const listItems = numbers.map((number) =>
        <li key={number.toString()}>
            {number}
        </li>
    )
    return (
        <ul>{listItems}</ul>
    )
}

const numbers = [1, 2, 3, 4, 5]
ReactDOM.render(
    <NumberList numbers={numbers} />,
    document.getElementById('root')
)

```

Exceptions where it is safe to use index as key

- If your list is static and will not change.
- The list will never be re-ordered.
- The list will not be filtered (adding/removing items from the list).
- There are no ids for the items in the list.

Note: Using index as a key can lead to potential unexpected behaviour within the component.

Q30. What is React Router? Why is switch keyword used in React Router v4?

React router implements a component-based approach to routing. It provides different routing components according to the needs of the application and platform. React Router keeps your UI in sync with the URL. It has a simple API with powerful features like lazy loading, dynamic route matching, and location transition handling built right in.

```

npm install react-router-dom
import React, { Component } from 'react'
import { Router, Route, Redirect, Switch } from 'react-router-dom'

```

```

import Todos from './components/Todos/Todos'
import TodosNew from './components/TodosNew/TodosNew'
import TodoShow from './components/TodoShow/TodoShow'

class Router extends Component {
  constructor(props) {
    super(props)
  }

  render() {
    return (
      <Router>
        <Switch>
          <Route path='/todos/new' component={ TodosNew } />
          <Route path='/todos/:id' component={ TodoShow } />
          <Route exact path='/' component={ Todos } />
          <Redirect from='*' to='/' />
        </Switch>
      </Router>
    )
  }
}

export default Router
< Router />

The < Router /> component wraps our main application routing. Nested within Router will be all of our < Route /> components, which will point to all other URLs.

<Switch />

The Switch component helps us to render the components only when path matches otherwise it fallbacks to the not found component. The <Switch> returns only one first matching route.

exact

The exact returns any number of routes that match exactly.

```

Q31. Explain the standard JavaScript toolchain, transpilation (via Babel or other compilers), JSX, and these items significance in recent development?

Typically, we use build tools like Grunt, Watchify/Browserify, Broccoli, or Webpack to watch the filesystem for file events (like when you add or edit a file). After this occurs, the build tool is configured to carry out a group of sequential or parallel tasks.

The rest of the tools belong in that group of sequential or parallel tasks:

- *Style linting* - typically a linter like JSCS is used to ensure the source code is following a certain structure and style
- *Dependency Management* - for JavaScript projects, most people use other packages from npm; some plugins exist for build systems (e.g. Webpack) and compilers (e.g. Babel) that allow automatic installation of packages being imported or require()'d
- *Transpilation* - a specific sub-genre of compilation, transpilation involves compiling code from one source version to another, only to a similar runtime level (e.g. ES6 to ES5)

- *Compilation* - specifically separate from transpiling ES6 and JSX to ES5, is the act of including assets, processing CSS files as JSON, or other mechanisms that can load and inject external assets and code into a file. In addition, there are all sorts of build steps that can analyze your code and even optimize it for you.
- *Minification and Compression* - typically part of – but not exclusively controlled by – compilation, is the act of minifying and compressing a JS file into fewer and/or smaller files
- *Source-Mapping* - another optional part of compilation is building source maps, which help identify the line in the original source code that corresponds with the line in the output code (i.e. where an error occurred)

Q32. How React handle or restrict Props to certain types?

React PropTypes are a good way to help you catching bugs by validating data types of values passed through props. They also offer possibilities to flag props as mandatory or set default values.

Example:

```
import React from 'react'
import PropTypes from 'prop-types'

const Person = (props) => <div>
  <h1>{props.firstName} {props.lastName}</h1>
  {props.country ? <p>Country: {props.country}</p> : null}
</div>

Person.propTypes = {
  firstName:PropTypes.string,
  lastName:PropTypes.string,
  country:PropTypes.string
}

export default Person
```

PropTypes define the type of a prop. So each time, a value is passed through a prop, it gets validated for its type. If you pass a value through a prop with a different data type than it is specified in the PropTypes, an error message will be printed in the console of your browser.

Q33. What is prop drilling and how can you avoid it?

React passes data to child components via props from top to bottom. While there are few props or child components, it is easy to manage and pass down data. But when the application grows, and want to pass data from the top level component to a 3rd or 4th level component but we end up passing these data to components on each level of the tree. This is called **Prop-drilling**.

Context API

The Context API solves some of these prop drilling problems. It let pass data to all of the components in the tree without writing them manually in each of them. Shared data can be anything: state, functions, objects, we name it, and it is accessible to all nested levels that are in the scope of the context.

Example:

```
import React from "react"
```

```

import ReactDOM from "react-dom"

// Create a Context
const NumberContext = React.createContext()
// It returns an object with 2 values:
// { Provider, Consumer }

function App() {
  // Use the Provider to make a value available to all
  // children and grandchildren
  return (
    <NumberContext.Provider value={10}>
      <div>
        <Display />
      </div>
    </NumberContext.Provider>
  )
}

function Display() {
  const value = useContext(NumberContext)
  return <div>The answer is {value}.</div>
}

```

Q34. If you wanted a component to perform an action only once when the component initially rendered - how would you achieve this in react?

The `componentDidMount()` lifecycle hook can be used with class components.

```

class Homepage extends React.Component {
  componentDidMount() {
    trackPageView('Homepage')
  }
  render() {
    return <div>Homepage</div>
  }
}

```

Any actions defined within a `componentDidMount()` lifecycle hook are called only once when the component is first mounted.

The `useEffect()` hook can be used with function components.

```

const Homepage = () => {
  useEffect(() => {
    trackPageView('Homepage')
  }, [])
}

return <div>Homepage</div>
}

```

The `useEffect()` hook is more flexible than the lifecycle methods used for class components. It receives two parameters:

- The first parameter it takes is a callback function to be executed.

- The optional second parameter it takes is an array containing any variables that are to be tracked.

The value passed as the second argument controls when the callback is executed:

- If the second parameter is undefined, the callback is executed every time that the component is rendered.
- If the second parameter contains an array of variables, then the callback will be executed as part of the first render cycle and will be executed again each time an item in the array is modified.
- If the second parameter contains an empty array, the callback will be executed only once as part of the first render cycle. The example above shows how passing an empty array can result in similar behaviour to the `componentDidMount()` hook within a function component.

Q35. How can automated tooling be used to improve the accessibility of a React application?

There are two main categories of automated tools that can be used to identify accessibility issues:

Static Analysis Tools

Linting tools like `ESLint` can be used with plugins such as `eslint-plugin-jsx-a11y` to analyse React projects at a component level. Static analysis tools run very quickly, so they bring a good benefit at a low cost.

Browser Tools

Browser accessibility tools such as `axe` and `Google Lighthouse` perform automated accessibility at the app level. This can discover more real-world issues, because a browser is used to simulate the way that a user interacts with a website.

Q36. What is the purpose of using super constructor with props argument?

The `super()` keyword is used to call the parent constructor. `super(props)` would pass `props` to the parent constructor.

```
class App extends React.Component {
  constructor(props) {
    super(props)
    this.state = {}
  }

  // React says we have to define render()
  render() {
    return <div>Hello world</div>
  }
}

export default App
```

Here, `super(props)` would call the `React.Component` constructor passing in `props` as the argument.

Q37. Why should not we update the state directly?

The `setState()` does not immediately mutate `this.state()` but creates a pending state transition. Accessing `this.state` after calling this method can potentially return the existing value.

There is no guarantee of synchronous operation of calls to `setState()` and calls may be batched for performance gains.

The `setState()` will always trigger a re-render unless conditional rendering logic is implemented in `shouldComponentUpdate()`. If mutable objects are being used and the logic cannot be implemented in `shouldComponentUpdate()`, calling `setState()` only when the new state differs from the previous state will avoid unnecessary re-renders.

Basically, if we modify `this.state()` directly, we create a situation where those modifications might get overwritten.

Example:

```
import React, { Component } from 'react'

class App extends Component {
  constructor(props) {
    super(props)

    this.state = {
      list: [
        { id: '1', age: 42 },
        { id: '2', age: 33 },
        { id: '3', age: 68 },
      ],
    }
  }

  onRemoveItem = id => {
    this.setState(state => {
      const list = state.list.filter(item => item.id !== id)

      return {
        list,
      }
    })
  }

  render() {
    return (
      <div>
        <ul>
          {this.state.list.map(item => (
            <li key={item.id}>
              The person is {item.age} years old.
              <button
                type="button"
                onClick={() => this.onRemoveItem(item.id)}
              >
                Remove
              </button>
            </li>
          ))
        </ul>
      </div>
    )
  }
}
```

```

        </li>
      )})
</ul>
</div>
)
}
}

export default App

```

Q38. What do these three dots (...) in React do?

The ES6 Spread operator or Rest Parameters is use to pass `props` to a React component. Let us take an example for a component that expects two props:

```
function App() {
  return <Hello firstName="Alex" lastName="K" />
}
```

Using the Spread operator, it become like this

```
function App() {
  const props = {firstName: 'Alex', lastName: 'K'}
  return <Hello {...props} />
}
```

When we use the `...props` syntax, actually it expand the `props` object from the parent component, which means all its attributes are passed down the child component that may not need them all. This will make things like debugging harder.

Using the Spread Operator with `setState()` for Setting the Nested State

let us suppose we have a state with a nested object in our component:

```
this.state = {
  stateObj: {
    attr1: '',
    attr2: '',
  },
}
```

We can use the Spread syntax to update the nested state object.

```
this.setState(state => ({
  person: {
    ...state.stateObj,
    attr1: 'value1',
    attr2: 'value2',
  },
}))
```

Q39. What are React Hooks? What are advantages of using React Hooks?

React Hooks are in-built functions that allow to use `state` and `lifecycle` methods inside functional components, they also work together with existing code, so they can easily be adopted into a codebase.

Rules of Hooks

- Make sure to not use Hooks inside loops, conditions, or nested functions
- Only use Hooks from inside React Functions

Built-in Hooks

Basic Hooks

- useState()
- useEffect()
- useContext()

Additional Hooks

- useReducer()
- useCallback()
- useMemo()
- useRef()
- useImperativeHandle()
- useLayoutEffect()
- useDebugValue()

React Hooks advantages

- Hooks are easier to work with and to test (as separated functions from React components*) and make the code look cleaner, easier to read — a related logic can be tightly coupled in a custom hook.
- Hooks allow to do by breaking the logic between components into small functions and using them inside the components.
- Improved code reuse
- Better code composition
- Better defaults
- Sharing non-visual logic with the use of custom hooks
- Flexibility in moving up and down the components tree.

Example: using classes

```
import React, { Component } from 'react'

class App extends Component {
  constructor(props) {
    super(props)

    this.state = {
      isButtonClicked: false,
    }
    this.handleClick = this.handleClick.bind(this)
  }

  handleClick() {
    this.setState((prevState) => ({
      isButtonClicked: !prevState.isButtonClicked,
    }))
  }
}
```

Example: using React Hooks

```
import React, { useState } from 'react'

const App = () => {
  const [isButtonClicked, setIsButtonClickedStatus] = useState(false)
```

```

        return (
            <button
                onClick={() => setIsButtonClickedStatus(!isButtonClicked) }
            >
                {isButtonClicked ? 'Clicked' : 'Click me, please'}
            </button>
        )
    }
}

```

Q40. How to apply validation on Props in React?

Props are an important mechanism for passing the **read-only** attributes to React components. React provides a way to validate the props using `PropTypes`. This is extremely useful to ensure that the components are used correctly.

```
npm install prop-types --save-dev
```

Example:

```

import React from 'react'
import PropTypes from 'prop-types'

App.defaultProps = {
    propBool: true,
    propArray: [1, 2, 3, 4, 5],
    propNumber: 100,
    propString: "Hello React!"
}

class App extends React.Component {

    render() {
        return (
            <fragment>
                <h3>Boolean: {this.props.propBool ? "True" : "False"}</h3>
                <h3>Array: {this.props.propArray}</h3>
                <h3>Number: {this.props.propNumber}</h3>
                <h3>String: {this.props.propString}</h3>
            </fragment>
        )
    }
}

App.propTypes = {
    propBool: PropTypes.bool.isRequired,
    propArray: PropTypes.array.isRequired,
    propNumber: PropTypes.number,
    propString: PropTypes.string,
}

export default App

```

Q41. What is the difference between using constructor vs `getInitialState` in React?

The two approaches are not interchangeable. You should initialize state in the `constructor()` when using ES6 classes, and define the `getInitialState()` method when using `React.createClass`.

```
import React from 'react'

class MyComponent extends React.Component {

  constructor(props) {
    super(props)
    this.state = { /* initial state */ }
  }
}
```

is equivalent to

```
var MyComponent = React.createClass({
  getInitialState() {
    return { /* initial state */ }
  },
})
```

The `getInitialState()` is used with `React.createClass` and `constructor()` is used with `React.Component`.

Q42. How to conditionally add attributes to React components?

Inline conditionals in attribute props

```
import React from 'react'

function App() {

  const [mood] = React.useState("happy")

  const greet = () => alert("Hi there! :)")

  return (
    <button onClick={greet} disabled={"happy" === mood ? false : true}>
      Say Hi
    </button>
  )
}
```

Q43. Do Hooks replace render props and higher-order components?

React Hooks

Hooks were designed to replace `class` and provide another great alternative to compose behavior into your components. Higher Order Components are also useful for composing behavior. Hooks encapsulate the functionality to easily reusable functions

```
const [active, setActive] = useState(defaultActive)
```

There are few build-in Hooks

```
import {
```

```

useState,
useReducer,
useEffect,
useCallback,
useMemo,
useRef,
...
} from 'react'

```

Higher Order Components

A Higher Order Component (HOC) is a component that takes a component and returns a component. HOCs are composable using point-free, declarative function composition.

Example: logger API

```

import React, { useEffect } from 'react'

const withLogging = Component => props => {
  useEffect(() => {
    fetch(`/logger?location=${ window.location }`)
  }, [])
  return <Component {...props} />
}
export default withLogging

```

To use it, you can mix it into an HOC that you'll wrap around every page:

```

import React from 'react'
import withAuth from './with-auth.js'
import withLogging from './with-logging.js'
import withLayout from './with-layout.js'

```

```

const page = compose(
  withRedux,
  withAuth,
  withLogging,
  withLayout('default'),
)

```

export default page

To use this for a page

```

import page from '../hocs/page.js'
import MyPageComponent from './my-page-component.js'

```

```

export default page(MyPageComponent)

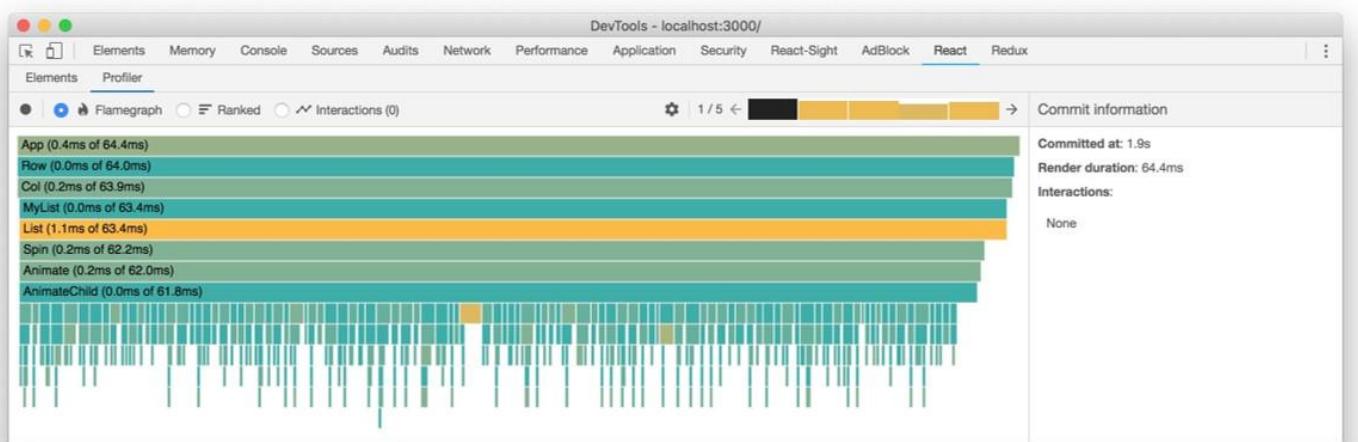
```

Q44. How to optimize React Performance?

React uses many techniques to minimize the number of DOM operations for us already. For many applications, if you are using the production build, you may already meet or surpass your performance expectations. Nevertheless, there are several ways you can speed up your application.

1. React DevTools Profiler

Experience performance problems with a specific component, the React DevTools Profiler is usually the first place to look.



2. Using `shouldComponentUpdate()` method

By default, React will render the virtual DOM and compare the difference for every component in the tree for any change in its props or state. But that is obviously not reasonable. As our app grows, attempting to re-render and compare the entire virtual DOM at every action will eventually slow the whole thing down.

React provides a simple lifecycle method to indicate if a component needs re-rendering and that is, `shouldComponentUpdate` which is triggered before the re-rendering process starts. The default implementation of this function returns true.

```
shouldComponentUpdate(nextProps, nextState) {
  return true
}
```

When this function returns true for any component, it allows the render differentiating process to be triggered. This gives us the power of controlling the render differentiating process. Suppose we need to prevent a component from being re-rendered, we need simply to return false from that function. As we can see from the implementation of the method, we can compare the current and next props and state to determine whether a re-render is necessary:

```
shouldComponentUpdate(nextProps, nextState) {
  return nextProps.id !== this.props.id
}
```

3. Using Pure Components

Pure Components in React are the components which do not re-renders when the value of `state` and `props` has been updated with the same values. If the value of the previous `state` or `props` and the new `state` or `props` is the same, the component is not re-rendered. Pure Components restricts the re-rendering ensuring the higher performance of the Component.

4. Using `React.memo`

`React.memo` is a higher order component. It's similar to `React.PureComponent` but for function components instead of classes.

```
const MyComponent = React.memo(function MyComponent(props) {
  /* render using props */
})
```

If your function component renders the same result given the same props, you can wrap it in a call to `React.memo` for a performance boost in some cases by memoizing the result. This means that React will skip rendering the component, and reuse the last rendered result.

`React.memo` only checks for prop changes. If your function component wrapped in `React.memo` has a `useState` or `useContext` Hook in its implementation, it will still rerender when `state` or `context` change.

5. Virtualizing Long Lists

In order to address the issue with our long chat feed, the React team recommends a technique called windowing. This technique only renders the portion of the list that is visible to the user (+/- a given offset) in order to reduce the time to render. As the user scrolls, new list items are retrieved and rendered. `react-window` and `react-virtualized` are two libraries that provide components to help with list virtualization.

[]

Q45. When would you use StrictMode component in React?

`StrictMode` is a tool for highlighting potential problems in an application. Like `Fragment`, `StrictMode` does not render any visible UI. It activates additional checks and warnings for its descendants. Strict mode checks are run in development mode only; they do not impact the production build.

```
import React from 'react'

export default function App() {
  return (
    <Fragment>
      <Header />
      <React.StrictMode>
        <div>
          <ComponentOne />
          <ComponentTwo />
        </div>
      </React.StrictMode>
      <Footer />
    </Fragment>
  )
}
```

In the above example, strict mode checks will not be run against the `<Header>` and `<Footer>` components. However, `<ComponentOne>` and `<ComponentTwo>`, as well as all of their descendants, will have the checks.

`React.StrictMode`, in order to be efficient and avoid potential problems by any side-effects, needs to trigger some methods and lifecycle hooks twice. These are:

- Class component `constructor()` method
- The `render()` method
- `setState()` updater functions (the first argument)
- The static `getDerivedStateFromProps()` lifecycle
- `React.useState()` function

Benefits of StrictMode

- 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

Q46. How does React renderer work exactly when we call setState?

The state allows React components to change their output over time in response to user actions, network responses, and anything else, without violating this rule. Components defined as classes have some additional features. Local state is a feature available only to class Components.

The `setState()` is the API method provided with the library so that the user is able to define and manipulate state over time. `setState()` is the only legitimate way to update state after the initial state setup.

Example:

```
import React, { Component } from 'react'

class Search extends Component {
  constructor(props) {
    super(props)

    this.state = {
      searchString: ''
    }
  }
}
```

we are passing an empty string as a value and, to update the state of `searchString`, we have to call `setState()`.

```
setState({ searchString: event.target.value })
```

Here, we are passing an object to `setState()`. The object contains the part of the state we want to update which, in this case, is the value of `searchString`. This is basically kicking off a process that React calls **reconciliation**. The reconciliation process is the way React updates the DOM, by making changes to the component based on the change in state.

When the request to `setState()` is triggered, React creates a new tree containing the reactive elements in the component (along with the updated state). This tree is used to figure out how the Search component's UI should change in response to the state change by comparing it with the elements of the previous tree.

Q47. How to avoid the need for binding in React?

1. Use Arrow Function in Class Property

Usually when we want to access this inside a class method we would need to bind it to method like so:

```
class Button extends Component {
  constructor(props) {
    super(props)
    this.state = { clicked: false }
  }
  handleClick = () => this.setState({ clicked: true })
  render() {
    return <button onClick={this.handleClick}>Click Me</button>
  }
}
```

```
}
```

Binding `this` to `handleClick()` in the `constructor()` allows us to use `this.setState()` from Component inside `handleClick()`.

2. Bind in Render

```
onChange={this.handleChange.bind(this)}
```

This approach is terse and clear, however, there are performance implications since the function is reallocated on every render.

3. Bind in Constructor

One way to avoid binding in render is to bind in the constructor

```
constructor(props) {
  super(props)
  this.handleChange = this.handleChange.bind(this)
}
```

This is the approach currently recommended in the React docs for "better performance in your application".

Q48. How does the state differ from props in React?

State

This is data maintained inside a component. It is local or owned by that specific component. The component itself will update the state using the `setState()` function.

Example:

```
class AppComponent extends React.Component {
  state = {
    msg : 'Hello World!'
  }

  render() {
    return <div>Message {this.state.msg}</div>
  }
}
```

Props

Data passed in from a parent component. `props` are read-only in the child component that receives them. However, callback functions can also be passed, which can be executed inside the child to initiate an update.

Example: The parent can pass a props by using this

```
<ChildComponent color='red' />
```

Inside the `ChildComponent` constructor we could access the props

```
class ChildComponent extends React.Component {
  constructor(props) {
    super(props)
    console.log(props.color)
  }
}
```

Props can be used to set the internal state based on a prop value in the constructor, like this

```
class ChildComponent extends React.Component {
  constructor(props) {
    super(props)
    this.state.colorName = props.color
  }
}
```

}

Props should never be changed in a child component. Props are also used to allow child components to access methods defined in the parent component. This is a good way to centralize managing the state in the parent component, and avoid children to have the need to have their own state.

Difference between State and Props

| Props | State |
|---|--|
| Props are read-only. | State changes can be asynchronous. |
| Props allow to pass data from one component to other components as an argument. | State holds information about the components. |
| Props can be accessed by the child component. | State cannot be accessed by child components. |
| Props are used to communicate between components. | States can be used for rendering dynamic changes with the component. |
| Stateless component can have Props. | Stateless components cannot have State. |
| Props are external and controlled by whatever renders the component. | The State is internal and controlled by the React Component itself. |

Q49. How would you create a form in React?

App.js

```
import React, { Component } from "react"
import countries from "./countries"
import './App.css'

export default function App() {
  const [email, setEmail] = React.useState("")
  const [password, setPassword] = React.useState("")
  const [country, setCountry] = React.useState("")
  const [acceptedTerms, setAcceptedTerms] = React.useState(false)

  const handleSubmit = (event) => {
    console.log(`Email: ${email}
    Password: ${password}
    Country: ${country}
    Accepted Terms: ${acceptedTerms}`)
  }
  event.preventDefault()
}

return (
  <form onSubmit={handleSubmit}>
    <h1>Create Account</h1>

    <label>
      Email:
      <input
        name="email"
```

```

        type="email"
        value={email}
        onChange={e => setEmail(e.target.value)}
        required />
    </label>

    <label>
        Password:
        <input
            name="password"
            type="password"
            value={password}
            onChange={e => setPassword(e.target.value)}
            required />
    </label>

    <label>
        Country:
        <select
            name="country"
            value={country}
            onChange={e => setCountry(e.target.value)}
            required>
            <option key=""></option>
            {countries.map(country => (
                <option key={country}>{country}</option>
            )))
        </select>
    </label>

    <label>
        <input
            name="acceptedTerms"
            type="checkbox"
            onChange={e => setAcceptedTerms(e.target.value)}
            required />
        I accept the terms of service
    </label>

        <button>Submit</button>
    </form>
)
}

```

App.css

```

* {
    box-sizing: border-box;
}

body {
    font-family: Lato;
    height: 97vh;
    width: 100%;
    display: flex;
    flex-direction: column;
    justify-content: center;
}

```

```
    align-items: center;
    background-color: #4A4E69;
}

form {
    display: flex;
    flex-direction: column;
    width: 400px;
    min-width: 100px;
    min-height: 400px;
    padding: 20px 40px 40px 40px;
    border-radius: 6px;
    box-shadow: 0px 8px 36px #222;
    background-color: #fefefe;
}

form > h1 {
    display: flex;
    justify-content: center;
    font-family: "Segoe UI", "Ubuntu", "Roboto", "Open Sans", "Helvetica Neue",
    sans-serif;
    font-size: 2em;
    font-weight: lighter;
    margin-top: 0.25em;
    color: #222;
    letter-spacing: 2px;
}

.info {
    padding-bottom: 1em;
    padding-left: 0.5em;
    padding-right: 0.5em;
}

label {
    margin-bottom: 0.5em;
    color: #444;
    font-weight: lighter;
}

input {
    display: flex;
    flex-direction: column;
    margin-bottom: 15px;
    width: 100%;
}
input, select {
    padding: 10px 10px;
    border-radius: 5px;
    border: 1px solid #d6d1d5;
    margin-top: 5px;
}
select {
    display: block;
    width: 100%;
```

```

    height: 35px;
}
input[type="checkbox"] {
    display: inline-block;
    width: auto;
    margin-top: 2em;
    margin-right: 10px;
}

button {
    min-width: 100%;
    cursor: pointer;
    margin-right: 0.25em;
    margin-top: 0.5em;
    padding:      0.938em;
    border: none;
    border-radius: 4px;
    background-color: #22223B;
    color: #fefefe;
}
button:hover {
    background-color: #4A4E69;
    color: #fefefe;
}

.error {
    color:#db2269;
    font-size: 0.5em;
    display: relative;
}

.submit {
    width: 100%;
    display: flex;
    flex-wrap: wrap;
}

```

Countries.js

```

export default [
    'Austria',
    'Denmark',
    'France',
    'Germany',
    'India',
    'Italy',
    'Poland',
    'Russia',
    'Sweden',
    'United States'
];

```

Output:

Create Account

Email:

Password:

Country:

I accept the terms of service

Submit

Q50. How to change the state of a child component from its parent in React?

Using Props

We will take two components, Parent and Child. And our Parent component will set the value depends on the Child Component. Child component holds the Input field and we are going to send the input field value to the Parent component.

```
function Parent() {
  const [value, setValue] = React.useState("")

  function handleChange(newValue) {
    setValue(newValue)
  }

  // We pass a callback to Child
  return <Child value={value} onChange={handleChange} />
}
```

As you see that we set the onChange property to the Child component. Next step is to create the Child component.

```
function Child(props) {
  function handleChange(event) {
    // Here, we invoke the callback with the new value
```

```

        props.onChange(event.target.value)
    }

    return <input value={props.value} onChange={handleChange} />
}

```

On the above codes, we have created function handleChange that will pass the value through props.onChange to our Parent component.

Q51. What do you understand with the term polling in React?

Using `setInterval()` inside React components allows us to execute a function or some code at specific intervals.

```

useEffect(() => {
  const interval = setInterval(() => {
    console.log('This will run every second!')
  }, 1000)
  return () => clearInterval(interval)
}, [])

```

The code above schedules a new interval to run every second inside of the `useEffect` Hook. This will schedule once the React component mounts for the first time. To properly clear the interval, we return `clearInterval()` from the `useEffect()` Hook, passing in the interval.

Using `setInterval` in React Components

To schedule a new interval, we call the `setInterval` method inside of a React component, like so:

```

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

const IntervalExample = () => {
  const [seconds, setSeconds] = useState(0)

  useEffect(() => {
    const interval = setInterval(() => {
      setSeconds(seconds => seconds + 1)
    }, 1000)
    return () => clearInterval(interval)
  }, [])

  return (
    <div className="App">
      <header className="App-header">
        {seconds} seconds have elapsed since mounting.
      </header>
    </div>
  )
}

export default IntervalExample

```

The example above shows a React component, `IntervalExample`, scheduling a new interval once it mounts to the DOM. The interval increments the seconds state value by one, every second.

Q52. What is the difference between Element, Component and Component instance in React?

A React Component is a template. A blueprint. A global definition. This can be either a function or a class (with a render function).

A React Element is what gets returned from components. It is an object that virtually describes the DOM nodes that a component represents. With a function component, this element is the object that the function returns. With a class component, the element is the object that the component's render function returns. React elements are not what we see in the browser. They are just objects in memory and we can not change anything about them.

Example:

```
import React from 'react'
import ReactDOM from 'react-dom'
import './index.css'

class MyComponent extends React.Component {
  constructor(props) {
    super(props)
    console.log('This is a component instance:', this)
  }

  render() {
    const another_element = <div>Hello, World!</div>
    console.log('This is also an element:', another_element)
    return another_element
  }
}

console.log('This is a component:', MyComponent)

const element = <MyComponent/>

console.log('This is an element:', element)

ReactDOM.render(
  element,
  document.getElementById('root')
)
```

React Elements

A React Element is just a plain old JavaScript Object without own methods. It has essentially four properties:

- `type`, a String representing an HTML tag or a reference referring to a React Component
- `key`, a String to uniquely identify an React Element
- `ref`, a reference to access either the underlying DOM node or React Component Instance)
- `props` (properties Object)

A React Element is not an instance of a React Component. It is just a simplified "description" of how the React Component Instance (or depending on the type an HTML tag) to be created should look like.

A React Element that describes a React Component doesn't know to which DOM node it is eventually rendered - this association is abstracted and will be resolved while rendering.

React Elements may contain child elements and thus are capable of forming element trees, which represent the Virtual DOM tree.

React Components and React Component Instances

A custom React Component is either created by `React.createClass` or by extending `React.Component` (ES2015). If a React Component is instantiated it expects a `props` Object and returns an instance, which is referred to as a React Component Instance.

A React Component can contain state and has access to the React Lifecycle methods. It must have at least a `render` method, which returns a React Element(-tree) when invoked. Please note that you never construct React Component Instances yourself but let React create it for you.

Q53. In which lifecycle event do you make AJAX requests in React?

According to official React docs, the recommended place to do Ajax requests is in `componentDidMount()` which is a lifecycle method that runs after the React component has been mounted to the DOM. This is so you can use `setState()` to update your component when the data is retrieved.

Example:

```
import React from 'react'

class MyComponent extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      error: null,
      isLoading: false,
      items: []
    }
  }

  componentDidMount() {
    fetch("https://api.example.com/items")
      .then(res => res.json())
      .then(
        (result) => {
          this.setState({
            isLoading: true,
            items: result.items
          })
        },
        // Note: it's important to handle errors here
        // instead of a catch() block so that we don't swallow
        // exceptions from actual bugs in components.
        (error) => {
          this.setState({
            isLoading: true,
            error
          })
        }
      )
  }

  render() {
```

```

const { error, isLoading, items } = this.state
if (error) {
  return <div>Error: {error.message}</div>
} else if (!isLoading) {
  return <div>Loading...</div>
} else {
  return (
    <ul>
      {items.map(item => (
        <li key={item.name}>
          {item.name} {item.price}
        </li>
      ))}
    </ul>
  )
}
}
}

```

Q54. What is meant by event handling in React?

Handling events with React elements is very similar to handling events on DOM elements. There are some syntax differences:

- React events are named using camelCase, rather than lowercase.
- With JSX you pass a function as the event handler, rather than a string.

```

class Toggle extends React.Component {
  constructor(props) {
    super(props)
    this.state = {isToggleOn: true}

    // This binding is necessary to make `this` work in the callback
    this.handleClick = this.handleClick.bind(this)
  }

  handleClick() {
    this.setState(state => ({
      isToggleOn: !state.isToggleOn
    }))
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    )
  }
}

ReactDOM.render(
  <Toggle />,
  document.getElementById('root')
)

```

Q55. How many outermost elements can be there in a JSX expression?

A JSX expression must have only one outer element. For Example:

```
const headings = (
  <div id = "outermost-element">
    <h1>I am a heading </h1>
    <h2>I am also a heading</h2>
  </div>
)
```

Q56. Explain DOM diffing?

Document Object Model

The DOM (Document Object Model) is an interface that represents an HTML document in a tree-like structure with nodes. This structure allows the document to be traversed and modified by programmers with each node being represented as an object. The DOM is created by the browser when a web page is loaded.

React's "Virtual DOM"

The "Virtual DOM" is very similar to the real DOM, in that it is a tree-like structure kept in-memory, where React elements are represented as objects. This tree has many of the same properties as the real DOM without the power to change what is on the screen. It is a javascript object representing components in your application which can be updated quickly and efficiently by React.

When a JSX element is rendered or the state of an element changes, a new Virtual DOM tree is created. The function responsible for the creation of this tree is React's render() function. This is a fast process because the virtual DOM tree is just a javascript object and the UI will not be repainted based on this new tree.

DOM Diffing

Once the Virtual DOM is created, React compares this new representation with a snapshot of the previous version of the virtual DOM to see exactly which elements have changed.

Once the difference is known, React updates only those objects that differ on the actual DOM and the browser re-paints the screen. The next time state or props changes for a component in the application, a new virtual DOM tree of React elements will be created and the process will repeat.

The process of checking the difference between the new Virtual DOM tree and the old Virtual DOM tree is called **diffing**. Diffing is accomplished by a **heuristic O(n)** algorithm. During this process, React will deduce the minimum number of steps needed to update the real DOM, eliminating unnecessary costly changes. This process is also referred to as **reconciliation**.

React implements a heuristic O(n) algorithm based on two assumptions:

1. Two elements of different types will produce different trees.
2. The developer can hint at which child elements may be stable across different renders with a key prop."

Q57. What does shouldComponentUpdate do and why is it important?

The `shouldComponentUpdate()` method allows Component to exit the Update life cycle if there is no reason to apply a new render. React does not deeply compare `props` by default. When `props` or `state` is updated React assumes we need to re-render the content.

The default implementation of this function returns true so to stop the re-render you need to return false here:

```
shouldComponentUpdate(nextProps, nextState) {
  console.log(nextProps, nextState)
  console.log(this.props, this.state)
  return false
}
```

Preventing unnecessary renders

The `shouldComponentUpdate()` method is the first real life cycle optimization method that we can leverage in React. It checks the current `props` and `state`, compares it to the next `props` and `state` and then returns true if they are different, or false if they are the same. This method is not called for the initial render or when `forceUpdate()` is used.

Q58. What is the purpose of `render()` function in React?

All React applications start at a root DOM node that marks the portion of the DOM that will be managed by React. When React is called to render the component tree it will first need the JSX in code to be converted into pure JavaScript. `render` function is part of the react component lifecycle where `ReactDOM` is the class object which exposes a method called `render` which is used to render the React JSX content into DOM.

Generally you would use `ReactDOM.render()` once in your App to render the top level component, all other components will be children to the top level component. A react component goes through a number of mounting and updating lifecycle method and decides to render the data in the `render` function. Any JSX code that you write in `render()` method is converted to

`React.createElement(tag, props, children)` before it is rendered into the DOM.

```
// App.js
import React from 'react'
import './App.css'

function App() {
  return (
    <div className="App">
      Hello World !
    </div>
  )
}

export default App
// index.js
import React from 'react'
import ReactDOM from 'react-dom'
import './index.css'
import App from './App/App'

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```

```
</React.StrictMode>,
document.getElementById('root')
)
```

Q59. What are React components?

Components are the building blocks of any React app and a typical React app will have many of these. Simply put, a component is a JavaScript class or function that optionally accepts inputs i.e. properties(`props`) and returns a React element that describes how a section of the UI (User Interface) should appear.

A React component can be either **stateful** or **stateless**. Stateful components are of the class type, while stateless components are of the function type.

Stateless Component

```
import React from 'react'

const ExampleComponent = (props) => {
    return (<h1>Welcome to React!</h1>)
}

export default class App extends React.Component {
    render() {
        return (
            <div>
                <ExampleComponent/>
            </div>
        )
    }
}
```

The above example shows a stateless component named `ExampleComponent` which is inserted in the `<App/>` component. The `ExampleComponent` just comprises of a `<h1>` element.

Stateful Component

```
import React from 'react'

class ExampleComponent extends React.Component {

    constructor(props) {
        super(props)
        this.state = {
            heading: "This is an Example Component!"
        }
    }

    render() {
        return (
            <div>
                <h1>{ this.props.welcomeMsg }</h1>
                <h2>{ this.state.heading }</h2>
            </div>
        )
    }
}

export default class App extends React.Component {
    render() {
```

```

const welcomeMsg = "Welcome to React!"
return (
  <div>
    <ExampleComponent welcomeMsg={welcomeMsg} />
  </div>
)
}
}

```

The above example shows a stateful component named `ExampleComponent` which is inserted in the `<App/>` component. The `ExampleComponent` contains a `<h1>` and the `<h2>` element wrapped in a `<div>`. The `<h1>` displays data using props while the `<h2>` takes its data from the internal state of the `ExampleComponent`.

Props

Props are an optional input, and can be used to send data to the component. They are immutable properties, which makes them read-only. This also makes them come in handy when you want to display fixed values.

State

A React component of class type maintains an internal state which acts as a data store for it. This state can be updated and whenever it is changed, React re-renders that component.

LifeCycle

Every component has **lifecycle methods**. They specify the behavior of the component when it undergoes a phase of its lifecycle.

Q60. How do I bind a function to a component instance?

There are several ways to make sure functions have access to component attributes like `this.props` and `this.state`, depending on which syntax and build steps you are using.

Bind in Constructor (ES5)

```

class App extends Component {
  constructor(props) {
    super(props)
    this.handleClick = this.handleClick.bind(this)
  }
  handleClick() {
    console.log('Click happened')
  }
  render() {
    return <button onClick={this.handleClick}>Click Me</button>
  }
}

```

Class Properties

```

class App extends Component {
  // Note: this syntax is experimental and not standardized yet.
  handleClick = () => {
    console.log('Click happened')
  }
  render() {
    return <button onClick={this.handleClick}>Click Me</button>
  }
}

```

Bind in Render

```

class App extends Component {
  handleClick() {
    console.log('Click happened')
  }
  render() {
    return <button onClick={this.handleClick.bind(this)}>Click Me</button>
  }
}

```

Note: Using `Function.prototype.bind` in `render` creates a new function each time the component renders, which may have performance implications

Arrow Function in Render

```

class App extends Component {
  handleClick() {
    console.log('Click happened')
  }
  render() {
    return <button onClick={() => this.handleClick()}>Click Me</button>
  }
}

```

Note: Using an arrow function in `render` creates a new function each time the component renders, which may break optimizations based on strict identity comparison.

Q61. How do I pass a parameter to an event handler or callback?

You can use an arrow function to wrap around an event handler and pass parameters:

```
<button onClick={() => this.handleClick(id)} />
```

This is equivalent to calling `.bind`

```
<button onClick={this.handleClick.bind(this, id)} />
```

Example: Passing params using arrow functions

```
const A = 65 // ASCII character code
```

```

class Alphabet extends React.Component {
  constructor(props) {
    super(props)
    this.handleClick = this.handleClick.bind(this)
    this.state = {
      justClicked: null,
      letters: Array.from({length: 26}, (_, i) => String.fromCharCode(A + i))
    }
  }
  handleClick(letter) {
    this.setState({ justClicked: letter })
  }
  render() {
    return (
      <div>
        Just clicked: {this.state.justClicked}
        <ul>
          {this.state.letters.map(letter =>
            <li key={letter} onClick={() => this.handleClick(letter)}>
              {letter}
            </li>
          )}
        </ul>
      </div>
    )
  }
}

```

```

        </li>
    ) }
</ul>
</div>
)
}
}

Example: Passing params using data-attributes
Alternately, you can use DOM APIs to store data needed for event handlers. Consider this
approach if you need to optimize a large number of elements or have a render tree that relies on
React.PureComponent equality checks.
const A = 65 // ASCII character code

class Alphabet extends React.Component {
  constructor(props) {
    super(props)
    this.handleClick = this.handleClick.bind(this)
    this.state = {
      justClicked: null,
      letters: Array.from({length: 26}, (_, i) => String.fromCharCode(A + i))
    }
  }

  handleClick(e) {
    this.setState({
      justClicked: e.target.dataset.letter
    })
  }

  render() {
    return (
      <div>
        Just clicked: {this.state.justClicked}
        <ul>
          {this.state.letters.map(letter =>
            <li key={letter} data-letter={letter} onClick={this.handleClick}>
              {letter}
            </li>
          )}
        </ul>
      </div>
    )
  }
}

```

Q62. How can I prevent a function from being called too quickly?

Throttle

Throttling prevents a function from being called more than once in a given window of time. The example below throttles a "click" handler to prevent calling it more than once per second.

```
import throttle from 'lodash.throttle'
```

```

class LoadMoreButton extends React.Component {
  constructor(props) {
    super(props)
    this.handleClick = this.handleClick.bind(this)
    this.handleClickThrottled = throttle(this.handleClick, 1000)
  }

  componentWillUnmount() {
    this.handleClickThrottled.cancel()
  }

  render() {
    return <button onClick={this.handleClickThrottled}>Load More</button>
  }

  handleClick() {
    this.props.loadMore()
  }
}

```

Debounce

Debouncing ensures that a function will not be executed until after a certain amount of time has passed since it was last called. This can be useful when you have to perform some expensive calculation in response to an event that might dispatch rapidly (eg scroll or keyboard events).

The example below debounces text input with a 250ms delay.

```

import debounce from 'lodash.debounce'

class Searchbox extends React.Component {
  constructor(props) {
    super(props)
    this.handleChange = this.handleChange.bind(this)
    this.emitChangeDebounced = debounce(this.emitChange, 250)
  }

  componentWillUnmount() {
    this.emitChangeDebounced.cancel()
  }

  render() {
    return (
      <input
        type="text"
        onChange={this.handleChange}
        placeholder="Search..."
        defaultValue={this.props.value}
      />
    )
  }

  handleChange(e) {
    // React pools events, so we read the value before debounce.
    // Alternately we could call `event.persist()` and pass the entire event.
    // For more info see reactjs.org/docs/events.html#event-pooling
    this.emitChangeDebounced(e.target.value)
  }
}

```

```

    }

    emitChange(value) {
      this.props.onChange(value)
    }
}

```

requestAnimationFrame throttling

`requestAnimationFrame` is a way of queuing a function to be executed in the browser at the optimal time for rendering performance. A function that is queued with `requestAnimationFrame` will fire in the next frame. The browser will work hard to ensure that there are 60 frames per second (60 fps). However, if the browser is unable to it will naturally limit the amount of frames in a second.

For example, a device might only be able to handle 30 fps and so you will only get 30 frames in that second. Using `requestAnimationFrame` for throttling is a useful technique in that it prevents you from doing more than 60 updates in a second. If you are doing 100 updates in a second this creates additional work for the browser that the user will not see anyway.

```

import rafSchedule from 'raf-schd'

class ScrollListener extends React.Component {
  constructor(props) {
    super(props)

    this.handleScroll = this.handleScroll.bind(this)

    // Create a new function to schedule updates.
    this.scheduleUpdate = rafSchedule(
      point => this.props.onScroll(point)
    )
  }

  handleScroll(e) {
    // When we receive a scroll event, schedule an update.
    // If we receive many updates within a frame, we'll only publish the
    latest value.
    this.scheduleUpdate({ x: e.clientX, y: e.clientY })
  }

  componentWillUnmount() {
    // Cancel any pending updates since we're unmounting.
    this.scheduleUpdate.cancel()
  }

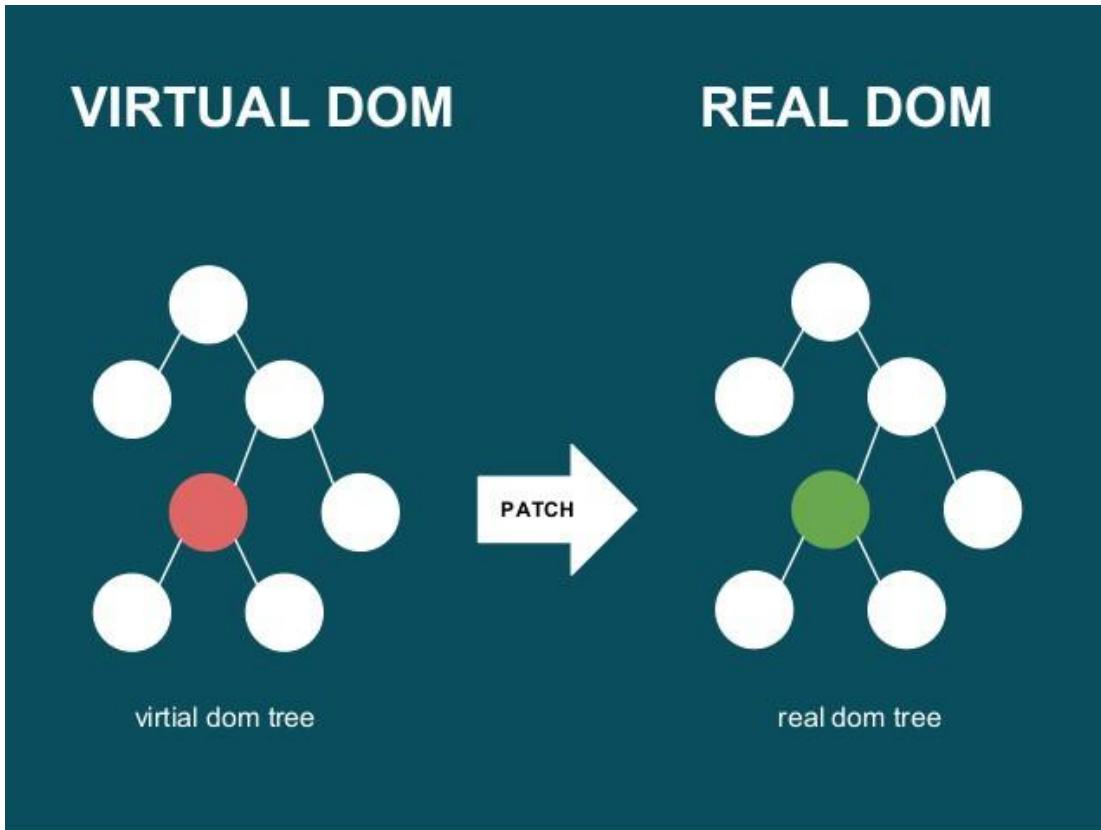
  render() {
    return (
      <div
        style={{ overflow: 'scroll' }}
        onScroll={this.handleScroll}
      >
        
      </div>
    )
  }
}

```

Q63. What is reconciliation in React?

Reconciliation is the process through which React updates the DOM.

As a developer we are creating tree of components, react then takes this tree, process it and we get a Virtual DOM that it's kept in memory. When there is an update in our application (e.g. change in state or props) react will take the updated Virtual DOM and compares it with the old one Virtual DOM, then decides what and how should be changed. This procedure is repeated all over again.



Also synced versions between Virtual DOM and "real" DOM are served by libraries such as **ReactDOM**. React needs to be very fast at comparing those trees, so it uses **heuristic algorithm** with complexity of **O(n)**, so this says for 1000 nodes we need 1000 comparasions. This approach is used instead of state of the art algorithms, which have complexity of **O(n^3)** => for 1000 nodes we need 1 billion comparasions.

Example: Let's build a simple component that adds two numbers. The numbers will be entered in an input field.

```
class App extends React.Component {  
  
  state = {  
    result: '',  
    entry1: '',  
    entry2: ''  
  }  
  
  handleEntry1 = (event) => {  
    this.setState({entry1: event.target.value})  
  }  
  
  handleEntry2 = (event) => {  
    this.setState({entry2: event.target.value})  
  }  
  
  calculateSum = () => {  
    const entry1 = this.state.entry1;  
    const entry2 = this.state.entry2;  
    if (entry1 === '' || entry2 === '') {  
      return ''  
    }  
    return Number(entry1) + Number(entry2);  
  }  
  
  render() {  
    return (  
      <div>  
        <h1>Calculator</h1>  
        <input type="text" value={this.state.entry1} onChange={this.handleEntry1}>  
        <input type="text" value={this.state.entry2} onChange={this.handleEntry2}>  
        <h2>Result: {this.calculateSum()}</h2>  
      </div>  
    );  
  }  
}
```

```

handleEntry2 = (event) => {
  this.setState({entry2: event.target.value})
}

handleAddition = (event) => {
  const firstInt = parseInt(this.state.entry1)
  const secondInt = parseInt(this.state.entry2)
  this.setState({result: firstInt + secondInt })
}

render() {
  const { entry1, entry2, result } = this.state
  return(
    <div>
      <div>
        Result: { result }
      </div>
      <span><input type='text' onChange={this.handleEntry1} /></span>
      <br />
      <br />
      <span><input type='text' onChange={this.handleEntry2} /></span>
      <div>
        <button onClick={this.handleAddition} type='submit'>Add</button>
      </div>
    </div>
  )
}
}

```

`ReactDOM.render(<App />, document.getElementById("root"))`

When an entry is made in the first input field, React creates a new tree. The new tree which is the virtual DOM will contain the new state for **entry1**. Then, React compares the virtual DOM with the old DOM and, from the comparison, it figures out the difference between both DOMs and makes an update to only the part that is different. A new tree is created each time the state of App component changes — when a value is entered in either of the inputs field, or when the button is clicked.

Q64. What are portals in React?

Portals provide a quick and seamless way to render children into a DOM node that exists outside the DOM hierarchy of the parent component.

Normally, a functional or a class component renders a tree of React elements (usually generated from JSX). The React element defines how the DOM of the parent component should look.

`ReactDOM.createPortal(child, container)`

Features

- It transports its children component into a new React portal which is appended by default to `document.body`.
- It can also target user specified DOM element.
- It supports server-side rendering
- It supports returning arrays (no wrapper div's needed)

- It uses `<Portal />` and `<PortalWithState />` so there is no compromise between flexibility and convenience.

When to use

The common use-cases of React portal include:

- Modals
- Tooltips
- Floating menus
- Widgets

Installation

```
npm install react-portal --save
```

Example: React Portal

```
// App.js
```

```
import React, {Component} from 'react'
import './App.css'
import PortalDemo from './PortalDemo.js'

class App extends Component {
  render () {
    return (
      <div className='App'>
        <PortalDemo />
      </div>
    )
  }
}

export default App
```

The next step is to create a portal component and import it in the App.js file.

```
// PortalDemo.js
```

```
import React from 'react'
import ReactDOM from 'react-dom'

function PortalDemo() {
  return ReactDOM.createPortal(
    <h1>Portals Demo</h1>,
    document.getElementById('portal-root')
  )
}
export default PortalDemo
```

Now, open the Index.html file and add a
element to access the child component outside the root node.

```
<!-- index.html -->

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta name="theme-color" content="#000000" />
```

```

<link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
<title>React App using Portal</title>
</head>
<body>
  <noscript>It is required to enable JavaScript to run this app.</noscript>
  <div id="root"></div>
  <div id="portal-root"></div>
</body>
</html>

```

Q65. What is ReactDOMServer?

The `ReactDOMServer` object enables you to render components to static markup. Typically, it's used on a Node server:

```

// ES modules
import ReactDOMServer from 'react-dom/server'
// CommonJS
var ReactDOMServer = require('react-dom/server')

```

The **Server-side rendering (SSR)** is a popular technique for rendering a client-side single page application (SPA) on the server and then sending a fully rendered page to the client. This allows for dynamic components to be served as static HTML markup.

- It allows your site to have a faster first page load time, which is the key to a good user experience
- This approach can be useful for search engine optimization (SEO) when indexing does not handle JavaScript properly.
- It is great when people share a page of your site on social media, as they can easily gather the metadata needed to nicely share the link (images, title, description..)

Example:

Creating an Express Server

```
npm install express
```

All the content inside the build folder is going to be served as-is, statically by Express.

```
// server/server.js
```

```

import path from 'path'
import fs from 'fs'

import express from 'express'
import React from 'react'
import ReactDOMServer from 'react-dom/server'

import App from '../src/App'

const PORT = 8080
const app = express()

const router = express.Router()

const serverRenderer = (req, res, next) => {
  fs.readFile(path.resolve('./build/index.html'), 'utf8', (err, data) => {
    if (err) {
      console.error(err)
      return res.status(500).send('An error occurred')
    }
  })
}

router.get('/', serverRenderer)

app.use(router)

app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`)
})

```

```

    }
    return res.send(
      data.replace(
        `<div id="root"></div>`,
        `<div id="root">${ReactDOMServer.renderToString(<App />)}</div>`
      )
    )
  })
}

router.use('/$', serverRenderer)

router.use(
  express.static(path.resolve(__dirname, '..', 'build'), { maxAge: '30d' })
)

// tell the app to use the above rules
app.use(router)

// app.use(express.static('./build'))
app.listen(PORT, () => {
  console.log(`SSR running on port ${PORT}`)
})

```

Now, in the client application, in your src/index.js, instead of calling `ReactDOM.render()`:
`ReactDOM.render(<App />, document.getElementById('root'))`
 call `ReactDOM.hydrate()`, which is the same but has the additional ability to attach event listeners to existing markup once React loads:

```
ReactDOM.hydrate(<App />, document.getElementById('root'))
```

All the Node.js code needs to be transpiled by Babel, as server-side Node.js code does not know anything about JSX, nor ES Modules (which we use for the include statements).

Babel Package

```
npm install @babel/register @babel/preset-env @babel/preset-react ignore-styles
```

Let's create an entry point in server/index.js:

```
require('ignore-styles')

require('@babel/register')({
  ignore: [/(node_modules)/],
  presets: ['@babel/preset-env', '@babel/preset-react']
})
```

```
require('../server')
```

Build the React application, so that the build/ folder is populated and run this:

```
# Build App
npm run build
```

```
# Run App on Express
node server/index.js
```

Q66. What will happen if you use props in initial state?

Using props to generate state in `getInitialState` often leads to duplication of "source of truth", i.e. where the real data is. This is because `getInitialState` is only invoked when the component is first created.

The danger is that if the `props` on the component are changed without the component being '*refreshed*', the new prop value will never be displayed because the constructor function (or `getInitialState`) will never update the current state of the component. The initialization of state from `props` only runs when the component is first created.

Bad

The below component won't display the updated input value

```
class App extends React.Component {  
  
    // constructor function (or getInitialState)  
    constructor(props) {  
        super(props)  
  
        this.state = {  
            records: [],  
            inputValue: this.props.inputValue  
        }  
    }  
  
    render() {  
        return <div>{this.state.inputValue}</div>  
    }  
}
```

Good

Using props inside render method will update the value:

```
class App extends React.Component {  
  
    // constructor function (or getInitialState)  
    constructor(props) {  
        super(props)  
  
        this.state = {  
            records: []  
        }  
    }  
  
    render() {  
        return <div>{this.props.inputValue}</div>  
    }  
}
```

Q67. How to re-render the view when the browser is resized?

Using React Hooks

```
import React, { useLayoutEffect, useState } from 'react'  
  
function useWindowSize() {  
  
    const [size, setSize] = useState([0, 0])
```

```

useLayoutEffect(() => {
  function updateSize() {
    setSize([window.innerWidth, window.innerHeight])
  }
  window.addEventListener('resize', updateSize)
  updateSize()
  return () => window.removeEventListener('resize', updateSize)
}, [])
}

return size
}

```

```

function ShowWindowDimensions(props) {

  const [width, height] = useWindowSize()
  return <span>Window size: {width} x {height}</span>
}

```

Using React classes

```

import React from 'react'

class ShowWindowDimensions extends React.Component {

  state = { width: 0, height: 0 }

  updateDimensions = () => {
    this.setState({ width: window.innerWidth, height: window.innerHeight })
  }
  /**
   * Add event listener
   */
  componentDidMount() {
    window.addEventListener('resize', this.updateDimensions)
  }
  /**
   * Remove event listener
   */
  componentWillUnmount() {
    window.removeEventListener('resize', this.updateDimensions)
  }

  render() {
    return (
      <span>Window size: {this.state.width} x {this.state.height}</span>
    )
  }
}

```

Q68. How can find the version of React at runtime in the browser?

Chrome Dev Tools

```
window.React.version
```

Q69. How to use https instead of http in create-react-app?

set the HTTPS environment variable to true, then start the dev server as usual with `npm start`:

```
# Windows  
set HTTPS=true&&npm start
```

Q70. Why is a component constructor called only once?

React's **reconciliation algorithm** assumes that without any information to the contrary, if a custom component appears in the same place on subsequent renders, it's the same component as before, so reuses the previous instance rather than creating a new one.

If you give each component a unique key prop, React can use the key change to infer that the component has actually been substituted and will create a new one from scratch, giving it the full component lifecycle.

```
renderContent() {  
  if (this.state.activeItem === 'item-one') {  
    return (  
      <Content title="First" key="first" />  
    )  
  } else {  
    return (  
      <Content title="Second" key="second" />  
    )  
  }  
}
```

Q71. Explain React Router 5 features?

React Router 5 embraces the power of hooks and has introduced four different hooks to help with routing.

```
<Route path="/">  
  <Home />  
</Route>
```

useHistory

- Provides access to the `history` prop in React Router
- Refers to the `history` package dependency that the router uses
- A primary use case would be for programmatic routing with functions, like `push`, `replace`, etc.

```
import { useHistory } from 'react-router-dom'  
  
function Home() {  
  const history = useHistory()  
  return <button onClick={() => history.push('/profile')}>Profile</button>  
}
```

useLocation

- Provides access to the `location` prop in React Router
- It is similar to `window.location` in the browser itself, but this is accessible everywhere as it represents the * Router state and location.
- A primary use case for this would be to access the query params or the complete route string.

```

import { useLocation } from 'react-router-dom'

function Profile() {
  const location = useLocation()
  useEffect(() => {
    const currentPath = location.pathname
    const searchParams = new URLSearchParams(location.search)
  }, [location])
  return <p>Profile</p>
}

```

useParams

- Provides access to search parameters in the URL
- This was possible earlier only using match.params.

```
import { useParams, Route } from 'react-router-dom'
```

```

function Profile() {
  const { name } = useParams()
  return <p>{name}'s Profile</p>
}

function Dashboard() {
  return (
    <>
      <nav>
        <Link to={`/profile/alex`}>Alex Profile</Link>
      </nav>
      <main>
        <Route path="/profile/:name">
          <Profile />
        </Route>
      </main>
    </>
  )
}

```

useRouteMatch

- Provides access to the match object
- If it is provided with no arguments, it returns the closest match in the component or its parents.
- A primary use case would be to construct nested paths.

```
import { useRouteMatch, Route } from 'react-router-dom'
```

```

function Auth() {
  const match = useRouteMatch()
  return (
    <>
      <Route path={`${match.url}/login`}*>
        <Login />
      </Route>
      <Route path={`${match.url}/register`}*>
        <Register />
      </Route>
    </>
  )
}

```

```
}
```

We can also use `useRouteMatch` to access a match without rendering a Route. This is done by passing it the location argument.

Redirect Component

The easiest way to use this method is by maintaining a redirect property inside the state of the component.

```
import { Redirect } from "react-router-dom"
...
state = { redirect: null }
render() {
  if (this.state.redirect) {
    return <Redirect to={this.state.redirect} />
  }
  return (
    // Your Code goes here
  )
}
```

History prop

Every component that is an immediate child of the `<Route>` component receives history object as a prop. This is the same history (library) which keeps history of the session of React Router. We can thus use its properties to navigate to the required paths.

```
this.props.history.push("/first")
```

Q72. What are the kinds of information that control a segment in React?

There are mainly two sorts of information that control a segment

- **State:** State information that will change, we need to utilize State.
- **Props:** Props are set by the parent and which are settled all through the lifetime of a part.

Q73. What are the drawbacks of MVW pattern?

MVW stands for **Model-View-Whatever**

- MVC - Model-View-Controller
- MVP - Model-View-Presenter
- MVVM - Model-View-ViewModel
- MVW / MV* / MVX - Model-View-Whatever
- HMVC - Hierarchical Model-View-Controller
- MMV - Multiuse Model View
- MVA - Model-View-Adapter

MVW is easy to manage in a simple application, with few models/controllers. But we can easily start to witness problems as we grow in size with the following problems:

1. There is need when models/controllers communicate with each others (through a service layer probably), and these modules changes the states of each others, and the more controllers, the more easy to lose control of who changed the state of a controller.

2. Asynchronous network calls to retrieve data add uncertainty of when the model will be changed or modified, and imagine the user changing the UI while a callback from asynchronous call comeback, then we will have "nondeterministic" status of the UI.
3. Change state/model has another layer of complexity which is the mutation. When to consider the state or model is changed and how to build tools to help recognize the mutation.
4. Adding to that if the application is a collaborative applications, (like google docs for examples) where lots of data changes happening in real-time.
5. No way to do undo (travel back in time) easily without adding so much extra code.

Q74. What is the difference between createElement and cloneElement?

JSX elements will be transpiled to `React.createElement()` functions to create React elements which are going to be used for the object representation of UI. Whereas `cloneElement` is used to clone an element and pass it new props.

The `React.cloneElement()` function returns a copy of a specified element. Additional props and children can be passed on in the function. We shoul use this function when a parent component wants to add or modify the props of its children.

```
import React from 'react'

export default class App extends React.Component {
  // rendering the parent and child component
  render() {
    return (
      <ParentComp>
        <MyButton/>
        <br/>
        <MyButton/>
      </ParentComp>
    )
  }
}

// The parent component
class ParentComp extends React.Component {
  render() {
    // The new prop to the added.
    let newProp = 'red'
    // Looping over the parent's entire children,
    // cloning each child, adding a new prop.
    return (
      <div>
        {React.Children.map(this.props.children,
          child => {
            return React.cloneElement(child,
              {newProp}, null)
          )))
      </div>
    )
  }
}
```

```
// The child component
class MyButton extends React.Component {
  render() {
    return <button style =
    {{ color: this.props.newProp }}>
      Hello World!</button>
  }
}
```

Q75. When should I be using `React.cloneElement` vs `this.props.children`?

The `React.cloneElement` only works if your child is a single React element.

Example:

```
<ReactCSSTransitionGroup
  component="div"
  transitionName="example"
  transitionEnterTimeout={500}
  transitionLeaveTimeout={500}
>
  {React.cloneElement(this.props.children, {
    key: this.props.location.pathname
  ))}
</ReactCSSTransitionGroup>
```

For almost everything `{this.props.children}` is used. Cloning is useful in some more advanced scenarios, where a parent sends in an element and the child component needs to change some props on that element or add things like `ref` for accessing the actual DOM element.

Example:

```
class Users extends React.Component {
  render() {
    return (
      <div>
        <h2>Users</h2>
        {this.props.children}
      </div>
    )
  }
}
```

Q76. Explain the Lists in React?

Using JSX we can show lists using JavaScript's built-in `Array.map()` method. The `.map()` method is often used to take one piece of data and convert it to another.

Keys are unique identifiers that must be attached to the top-level element inside a map. Keys are used by React to know how to update a list whether adding, updating, or deleting items. This is part of how React is so fast with large lists.

Example: Rendering an Array of Objects as a List

```
import React, { Component } from "react"

class Item extends Component {
  state = {
```

```

lists: [
  {
    id: 0,
    context: "Success",
    modifier: "list-group-item list-group-item-success"
  },
  {
    id: 1,
    context: "Warning",
    modifier: "list-group-item list-group-item-warning"
  },
  {
    id: 2,
    context: "Danger",
    modifier: "list-group-item list-group-item-danger"
  }
]
}

render() {
  return (
    <React.Fragment>
      <ul className = "list-group">
        {this.state.lists.map(list => (
          <li key = {list.id} className = {list.modifier}>
            {list.context}
          </li>
        )))
      </ul>
    </React.Fragment>
  )
}
}

export default Item

```

Q77. Why is it necessary to start component names with a capital letter?

In JSX, lower-case tag names are considered to be HTML tags. However, lower-case tag names with a dot (property accessor) aren't.

When an element type starts with a lowercase letter, it refers to a built-in component like or and results in a string `<div>` or `` passed to `React.createElement`. Types that start with a capital letter like compile to `React.createElement('Foo')` and correspond to a component defined or imported in your JavaScript file.

- `<component />` compiles to `React.createElement('component')` (html tag)
- `<Component />` compiles to `React.createElement(Component)`
- `<obj.component />` compiles to `React.createElement(obj.component)`

Q78. What are fragments? Why are fragments better than container divs?

Fragments allows to group a list of children without adding extra nodes to the DOM.

Example:

```
class App extends React.Component {  
  render() {  
    return (  
      <React.Fragment>  
        <ChildA />  
        <ChildB />  
        <ChildC />  
      </React.Fragment>  
    )  
  }  
}
```

Benefits

- It's a tiny bit faster and has less memory usage (no need to create an extra DOM node). This only has a real benefit on very large and/or deep trees, but application performance often suffers from death by a thousand cuts. This is one cut less.
- Some CSS mechanisms like Flexbox and CSS Grid have a special parent-child relationship, and adding divs in the middle makes it hard to keep the desired layout while extracting logical components.
- The DOM inspector is less cluttered.

Q79. What is Forwarding Refs in React?

Ref forwarding is a technique for passing a `ref` through a component to one of its children. It is very useful for cases like reusable component libraries and Higher Order Components (HOC).

We can forward a `ref` to a component by using the `React.forwardRef()` function. Ref forwarding allows components to take a ref they receive and pass it further down (in other words, "forward" it) to a child.

Example:

```
// Ref.js  
const TextInput = React.forwardRef((props, ref) => (  
  <input type="text" placeholder="Hello World" ref={ref} />  
)  
  
const inputRef = React.createRef()  
  
class CustomTextInput extends React.Component {  
  handleSubmit = e => {  
    e.preventDefault()  
    console.log(inputRef.current.value)  
  }  
  
  render() {  
    return (  
      <div>  
        <form onSubmit={e => this.handleSubmit(e)}>  
          <TextInput ref={inputRef} />  
        </form>  
      </div>  
    )  
  }  
}
```

```

        <button>Submit</button>
    </form>
</div>
)
}
}

```

In the example above, we have a component called TextInput that has a child which is an input field. First, we start by creating a ref with the line of code below:

```
const inputRef = React.createRef()
```

We pass our ref down to `<TextInput ref={inputRef}>` by specifying it as a JSX attribute. React then forwards the `ref` to the `forwardRef()` function as a second argument. Next, We forward this `ref` argument down to `<input ref={ref}>`. The value of the DOM node can now be accessed at `inputRef.current`.

Q80. Which is the preferred option callback refs or findDOMNode()?

It is preferred to use **callback refs** over `findDOMNode()` API. Because `findDOMNode()` prevents certain improvements in React in the future.

The legacy approach of using `findDOMNode()`:

```
class MyComponent extends Component {
  componentDidMount() {
    findDOMNode(this).scrollIntoView()
  }

  render() {
    return <div />
  }
}
```

The recommended approach is:

```
class MyComponent extends Component {
  componentDidMount() {
    this.node.scrollIntoView()
  }

  render() {
    return <div ref={node => this.node = node} />
  }
}
```

Q81. How is React Router different from Conventional Routing?

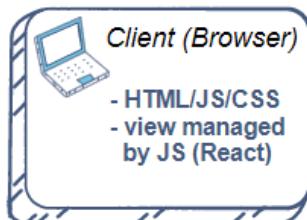
Old Architecture

Future requests send/receive all views/assets

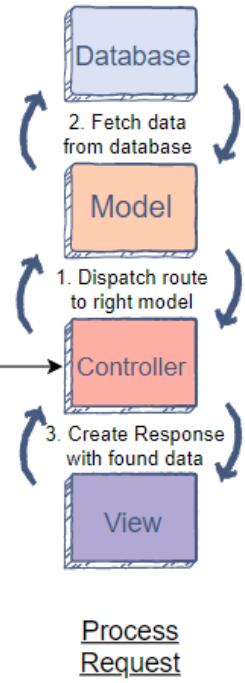


GET http://wikipedia.org

Server



GET http://educative.io



Future requests are just for data

Modern Architecture

In React, there is only a single 'Html' file involved. Whenever a user types in a new URL request, instead of fetching data from the server, the Router swaps in a different Component for each new URL request. The user is tricked into switching among multiple pages but in reality, each separate Component re-renders achieving multiple views as per our needs.

How does React achieve this?

In React, the Router looks at the **History** of each Component and when there is any change in the History, that Component re-renders. Until Router version 4 we had to manually set the History value. However, from Router v4 the base path is bypassed by the `<BrowserRouter>` saving us a lot of work.

Q82. How many ways can we style the React Component?

1. CSS Stylesheet

```
.DottedBox {  
  margin: 40px;  
  border: 5px dotted pink;  
}  
  
.DottedBox_content {  
  font-size: 15px;  
  text-align: center;  
}  
  
import React from 'react'  
import './DottedBox.css'  
  
const DottedBox = () => (  
  <div className="DottedBox">  
    <p className="DottedBox_content">Get started with CSS styling</p>  
  </div>
```

```

)
export default DottedBox

```

2. Inline styling

In React, inline styles are not specified as a string. Instead they are specified with an object whose **key** is the **camelCased** version of the style name, and whose value is the style's value, usually a string.

```
import React from 'react'
```

```

const divStyle = {
  margin: '40px',
  border: '5px solid pink'
}
const pStyle = {
  fontSize: '15px',
  textAlign: 'center'
}

const Box = () => (
  <div style={divStyle}>
    <p style={pStyle}>Get started with inline style</p>
  </div>
)

```

```
export default Box
```

- We can create a variable that stores style properties and then pass it to the element like `style={nameOfvariable}`
- We can also pass the styling directly `style={{color: 'pink'}}`

3. CSS Modules

A CSS Module is a CSS file in which all class names and animation names are scoped locally by default.

```

:local(.container) {
  margin: 40px;
  border: 5px dashed pink;
}
:local(.content) {
  font-size: 15px;
  text-align: center;
}
import React from 'react'
import styles from './DashedBox.css'

const DashedBox = () => (
  <div className={styles.container}>
    <p className={styles.content}>Get started with CSS Modules style</p>
  </div>
)

export default DashedBox

```

we import css file `import styles './DashedBox.css'`, then we access to `className` as we access to object.

- `:local(.className)` - this when you use `create-react-app` because of webpack configurations
- `.className` - this if you use your own react boilerplate.

4. Styled-components

Styled-components is a library for React and React Native that allows to use component-level styles in component application that are written with a mixture of JavaScript and CSS

```
npm install styled-components --save
import React from 'react'
import styled from 'styled-components'

const Div = styled.div`
  margin: 40px;
  border: 5px outset pink;
  &:hover {
    background-color: yellow;
  }
`;

const Paragraph = styled.p`
  font-size: 15px;
  text-align: center;
`;

const OutsetBox = () => (
  <Div>
    <Paragraph>Get started with styled-components</Paragraph>
  </Div>
)

export default OutsetBox
```

Q83. What are the advantages of using jsx?

JSX is an optional syntax extension to JavaScript that makes writing your own components much easier. It accepts HTML quoting and makes a subcomponent rendering easier. In fact, it is a set of shortcuts for writing `React.createElement` with a few rules to make your source cleaner and simpler.

While JSX is often a matter of dispute, it can prove useful in building high-volume apps or custom components, excluding typos in large tree structures, and making it easier to convert from HTML mockups to ReactElement trees. Besides that, it provides React developers with informative warning and error messages and also helps to prevent code injections.

Q84. What are the popular animation package in React?

ReactCSSTransitionGroup

`ReactCSSTransitionGroup` is a high-level API based on `ReactTransitionGroup` and is an easy way to perform CSS transitions and animations when a React component enters or leaves the DOM. It has four components that display transitions from one component state to another using a declarative API used for mounting and unmounting of components:

1. Transition

2. CSSTransition
3. SwitchTransition
4. TransitionGroup

Example:

```
import ReactCSSTransitionGroup from 'react-transition-group'

class TodoList extends React.Component {
  constructor(props) {
    super(props)
    this.state = {items: ['hello', 'world', 'click', 'me']}
    this.handleAdd = this.handleAdd.bind(this)
  }

  handleAdd() {
    const newItems = this.state.items.concat([
      prompt('Enter some text')
    ])
    this.setState({items: newItems})
  }

  handleRemove(i) {
    let newItems = this.state.items.slice()
    newItems.splice(i, 1)
    this.setState({items: newItems})
  }

  render() {
    const items = this.state.items.map((item, i) => (
      <div key={item} onClick={() => this.handleRemove(i)}>
        {item}
      </div>
    ))
  }

  return (
    <div>
      <button onClick={this.handleAdd}>Add Item</button>
      <ReactCSSTransitionGroup
        transitionName="example"
        transitionEnterTimeout={500}
        transitionLeaveTimeout={300}>
        {items}
      </ReactCSSTransitionGroup>
    </div>
  )
}
}
```

In this component, when a new item is added to ReactCSSTransitionGroup it will get the `example-enter` CSS class and the `example-enter-active` CSS class added in the next tick. This is a convention based on the `transitionName` prop.

Q85. Explain synthetic event in React js?

Inside React event handlers, the event object is wrapped in a `SyntheticEvent` object. These objects are pooled, which means that the objects received at an event handler will be reused for other events to increase performance. This also means that accessing the event object's properties asynchronously will be impossible since the event's properties have been reset due to reuse.

The following piece of code will log null because event has been reused inside the `SyntheticEvent` pool:

```
function handleClick(event) {
  setTimeout(function () {
    console.log(event.target.name)
  }, 1000)
}
```

To avoid this we need to store the event's property:

```
function handleClick(event) {
  let name = event.target.name
  setTimeout(function () {
    console.log(name)
  }, 1000)
}
```

SyntheticEvent Object

```
void preventDefault()
void stopPropagation()
boolean isPropagationStopped()
boolean isDefaultPrevented()
void persist()
boolean bubbles
boolean cancelable
DOMEVENTTARGET currentTarget
boolean defaultPrevented
number eventPhase
boolean isTrusted
DOMEVENT nativeEvent
DOMEVENTTARGET target
number timeStamp
string type
```

Q86. What is Event Pooling in React?

The `SyntheticEvent` is pooled. This means that the `SyntheticEvent` object will be reused and all properties will be nullified after the event callback has been invoked. This is for performance reasons. As such, you cannot access the event in an asynchronous way.

Example:

```
function onClick(event) {
  console.log(event) // => nullified object.
  console.log(event.type) // => "click"
  const eventType = event.type // => "click"

  setTimeout(function() {
    console.log(event.type) // => null
    console.log(eventType) // => "click"
  }, 0)

  // Won't work. this.state.clickEvent will only contain null values.
```

```

this.setState({clickEvent: event})

// You can still export event properties.
this.setState({eventType: event.type})
}

```

If we want to access the event properties in an asynchronous way, we should call `event.persist()` on the event, which will remove the synthetic event from the pool and allow references to the event to be retained by user code.

Q87. What are error boundaries in React?

Error boundaries are React components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of the component tree that crashed. Error boundaries catch errors during rendering, in lifecycle methods, and in constructors of the whole tree below them.

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

Example:

```

import React, {Component} from 'react'

class ErrorBoundary extends Component {
  state = {
    isErrorOccured: false,
    errorMessage: ''
  }
  componentDidCatch = (error, info) => {
    this.setState({
      isErrorOccured: true,
      errorMessage: error
    })
  }
  render() {
    if(this.state.isErrorOccured) {
      return <p>Something went wrong</p>
    } else {
      return <div>{this.props.children}</div>
    }
  }
}

export default ErrorBoundary

```

Here, We have a state object having two variables `isErrorOccured` and `errorMessage` which will be updated to true if any error occurs. We have used a React life cycle method `componentDidCatch` which receives two arguments `error` and `info` related to it.

How to use error boundary

```

<ErrorBoundary>
  <User/>
</ErrorBoundary>

```

Error boundaries do not catch errors for:

- Event handlers
- Asynchronous code (e.g. setTimeout())
- Server side rendering
- Errors thrown in the error boundary itself

Q88. How can you re-render a component without using setState() function?

React components automatically re-render whenever there is a change in their state or props. A simple update of the state, from anywhere in the code, causes all the User Interface (UI) elements to be re-rendered automatically.

However, there may be cases where the render() method depends on some other data. After the initial mounting of components, a re-render will occur.

1. Using setState()

In the following example, the `setState()` method is called each time a character is entered into the text box. This causes re-rendering, which updates the text on the screen.

```
import React, { Component } from 'react'
import 'bootstrap/dist/css/bootstrap.css'

class Greeting extends Component {
  state = {
    fullname: '',
  }

  stateChange = (f) => {
    const {name, value} = f.target
    this.setState({
      [name]: value,
    })
  }

  render() {
    return (
      <div className="text-center">
        <label htmlFor="fullname"> Full Name: </label>
        <input type="text" name="fullname" onChange={this.stateChange} />
        <div className="border border-primary py-3">
          <h4> Greetings, {this.state.fullname}!</h4>
        </div>
      </div>
    )
  }
}

export default Greeting
```

2. Using forceUpdate()

The following example generates a random number whenever it loads. Upon clicking the button, the `forceUpdate()` function is called which causes a new, random number to be rendered:

```
import React, { Component } from 'react'

class App extends React.Component{
```

```

constructor() {
  super()
  this.forceUpdateHandler = this.forceUpdateHandler.bind(this)
}

forceUpdateHandler() {
  this.forceUpdate()
}

render() {
  return (
    <div>
      <button onClick={this.forceUpdateHandler}>FORCE UPDATE</button>
      <h4>Random Number: { Math.random() }</h4>
    </div>
  )
}
}

export default App

```

Note: We should try to avoid all uses of forceUpdate() and only read from this.props and this.state in render().

Q89. What is difference between componentDidMount() and componentWillMount()?

componentDidMount()

The `componentDidMount()` is executed after the first render only on the client side. This is where AJAX requests and DOM or state updates should occur. This method is also used for integration with other JavaScript frameworks and any functions with delayed execution such as `setTimeout()` or `setInterval()`.

Example:

```

import React, { Component } from 'react'

class App extends Component {

  constructor(props) {
    super(props)
    this.state = {
      data: 'Alex Belfort'
    }
  }

  getData() {
    setTimeout(() => {
      console.log('Our data is fetched')
      this.setState({
        data: 'Hello Alex'
      })
    }, 1000)
  }
}

```

```

componentDidMount() {
  this.getData()
}

render() {
  return (
    <div>
      {this.state.data}
    </div>
  )
}
}

```

`export default App`

componentWillMount()

The `componentWillMount()` method is executed before rendering, on both the server and the client side. `componentWillMount()` method is the least used lifecycle method and called before any HTML element is rendered. It is useful when we want to do something programatically right before the component mounts.

Example:

```

import React, { Component } from 'react'

class App extends Component {

  constructor(props) {
    super(props)
    this.state = {
      data: 'Alex Belfort'
    }
  }

  componentWillMount() {
    console.log('First this called')
  }

  getData() {
    setTimeout(() => {
      console.log('Our data is fetched')
      this.setState({
        data: 'Hello Alex'
      })
    }, 1000)
  }

  componentDidMount() {
    this.getData()
  }

  render() {
    return (
      <div>
        {this.state.data}
      </div>
    )
  }
}

```

```
}

export default App
```

Q90. Explain the use of Webpack and Babel in React?

Babel

Babel is a JS transpiler that converts new JS code into old ones. It is a very flexible tool in terms of transpiling. One can easily add presets such as `es2015`, `es2016`, `es2017`, or `env`; so that Babel compiles them to ES5. Babel allows us to have a clean, maintainable code using the latest JS specifications without needing to worry about browser support.

Webpack

Webpack is a modular build tool that has two sets of functionality — Loaders and Plugins. Loaders transform the source code of a module. For example, `style-loader` adds CSS to DOM using style tags. `sass-loader` compiles SASS files to CSS. `babel-loader` transpiles JS code given the presets. Plugins are the core of Webpack. They can do things that loaders can't. For example, there is a plugin called `UglifyJS` that minifies and uglifies the output of webpack.

create-react-app

[create-react-app](#), a popular tool that lets you set up a React app with just one command. You don't need to get your hands dirty with Webpack or Babel because everything is preconfigured and hidden away from you.

Example: Quick Start

```
npx create-react-app my-app
cd my-app
npm start
```

Q91. Why to avoid using `setState()` after a component has been unmounted?

Calling `setState()` after a component has unmounted will emit a warning. The "setState warning" exists to help you catch bugs, because calling `setState()` on an unmounted component is an indication that your app/component has somehow failed to clean up properly. Specifically, calling `setState()` in an unmounted component means that your app is still holding a reference to the component after the component has been unmounted - which often indicates a memory leak.

Example:

```
class News extends Component {
  _isMounted = false // flag to check Mounted

  constructor(props) {
    super(props)

    this.state = {
      news: [],
    }
  }

  componentDidMount() {
    this._isMounted = true
```

```

axios
  .get('https://hn.algolia.com/api/v1/search?query=react')
  .then(result => {
    if (this._isMounted) {
      this.setState({
        news: result.data.hits,
      })
    }
  })
}

componentWillUnmount() {
  this._isMounted = false
}

render() {
  return (
    <ul>
      {this.state.news.map(topic => (
        <li key={topic.objectID}>{topic.title}</li>
      )))
    </ul>
  )
}
}
}
}

```

Here, even though the component got unmounted and the request resolves eventually, the flag in component will prevent to set the state of the React component after it got unmounted.

Q92. How to set focus on an input field after rendering?

Refs can be used to access DOM nodes or React components that are rendered in the render method. Refs are created with `React.createRef()` function. Refs can then be assigned to an element with ref-attribute. Following example shows a component that will focus to the text input when rendered.

```

class AutoFocusTextInput extends React.Component {

  constructor(props) {
    super(props)
    this.textInput = React.createRef()
  }
  componentDidMount() {
    this.textInput.current.focus()
  }
  render() {
    return <input ref={this.textInput} />
  }
}

```

Q93. How do you set a timer to update every second?

Using `setInterval()` inside React components allows us to execute a function or some code at specific intervals. A function or block of code that is bound to an interval executes until it is stopped. To stop an interval, we can use the `clearInterval()` method.

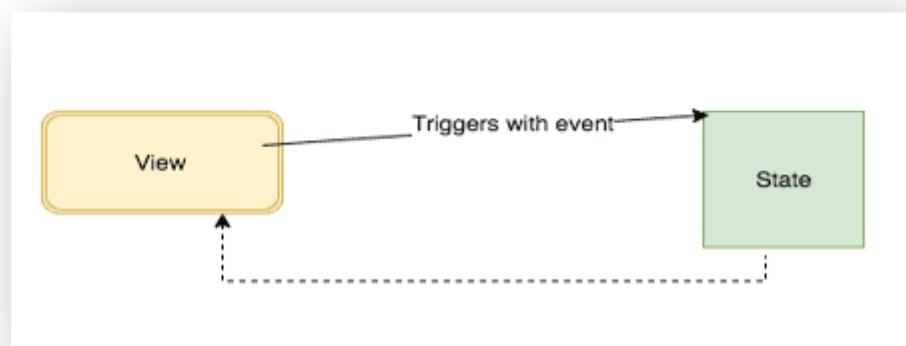
Example:

```
class Clock extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      time: new Date().toLocaleString()
    }
  }
  componentDidMount() {
    this.intervalID = setInterval(
      () => this.tick(),
      1000
    )
  }
  componentWillUnmount() {
    clearInterval(this.intervalID)
  }
  tick() {
    this.setState({
      time: new Date().toLocaleString()
    })
  }
  render() {
    return (
      <p className="App-clock">
        The time is {this.state.time}.
      </p>
    )
  }
}
```

Q94. How to implement two way data binding in React js?

Two data binding means

- The data we changed in the view has updated the state.
- The data in the state has updated the view.



```
class UserInput extends React.Component{
```

```

state = {
  name: "Hello React"
}

handleChange = (e) =>{
  this.setState({
    name: e.target.value
  })
}

render() {
  return(
    <div>
      <h1>{this.state.name}</h1>
      <input type="text"
        onChange={this.handleChange}
        value={this.state.name} />
    </div>
  )
}
}

```

In the above code, we have attached an `onChange()` event handler to the input element and also `value` attribute is connected to the `this.state.name` so that the `value` attribute is always synced with `this.state.name` property.

Whenever the user changes an input in the view it triggers the `onChange` event handler then it calls the `this.setState` method and updates the `this.state.name` property value at final the `UserInput` component is re-rendered with the updated changes.

This is also called **controlled component** because the value of an input element is controlled by the react.

Two way data binding in React hooks

```

import React, {useState} "react"

function App() {
  const [name, setName] = useState('')

  const handleChange = (e) => {
    setName(e.target.value)
  }

  return (
    <div>
      <input onChange={handleChange} value={name} />
      <h1>{name}</h1>
    </div>
  )
}

export default App

```

Q95. How to show and hide elements in React

Returning Null

```

const AddToCart = ({ available }) => {
  if (!available) return null

  return (
    <div className="full tr">
      <button className="product--cart-button">Add to Cart</button>
    </div>
  )
}

```

Ternary Display

When you need to control whether one element vs. another is displayed, or even one element vs. nothing at all (null), you can use the ternary operator embedded inside of a larger portion of JSX.

```

<div className="half">
  <p>{description}</p>

  {remaining === 0 ? (
    <span className="product-sold-out">Sold Out</span>
  ) : (
    <span className="product-remaining">{remaining} remaining</span>
  )}
</div>

```

In this case, if there are no products remaining, we will display "Sold Out"; otherwise we will display the number of products remaining.

Shortcut Display

It involves using a conditional inside of your JSX that looks like `checkIfTrue && display if true`. Because if statements that use `&&` operands stop as soon as they find the first value that evaluates to false, it won't reach the right side (the JSX) if the left side of the equation evaluates to false.

```

<h2>
  <span className="product--title__large">{nameFirst}</span>
  {nameRest.length > 0 && (
    <span className="product--title__small">{nameRest.join(" ") }</span>
  )}
</h2>

```

Using Style Property

```
<div style={{ display: showInfo ? "block" : "none" }}>info</div>
```

Q96. How to trigger click event programmatically?

We can use `ref` prop to acquire a reference to the underlying `HTMLInputElement` object through a callback, store the reference as a class property, then use that reference to later trigger a click from your event handlers using the `HTMLElement.click` method.

Example:

```

class MyComponent extends React.Component {

  render() {
    return (
      <div onClick={this.handleClick}>
        <input ref={input => this.inputElement = input} />
      </div>
    )
  }
}

MyComponent.handleClick = () => {
  const input = this.inputElement
  if (input) {
    input.click()
  }
}

```

```

    }

handleClick = (e) => {
  this.inputElement.click()
}

}

```

Note: The ES6 arrow function provides the correct lexical scope for this in the callback.

Q97. How to display style based on props value?

```

import styled from 'styled-components'

const Button = styled.button`
background: ${props => props.primary ? 'palevioletred' : 'white'}
color: ${props => props.primary ? 'white' : 'palevioletred'}
`;

function MyPureComponent(props) {
  return (
    <div>
      <Button>Normal</Button>
      <Button primary>Primary</Button>
    </div>
  )
}

```

Q98. How to convert text to uppercase on user input entered?

```

import React, { useState } from "react"
import ReactDOM from "react-dom"

const toInputUppercase = e => {
  e.target.value = ("" + e.target.value).toUpperCase()
}

const App = () => {
  const [name, setName] = useState("")

  return (
    <input
      name={name}
      onChange={e => setName(e.target.value)}
      onInput={toInputUppercase} // apply on input which do you want to be
capitalized
    />
  )
}

ReactDOM.render(<App />, document.getElementById("root"))

```

Q99. How to create props proxy for HOC component?

It's nothing more than a function, `propsProxyHOC`, that receives a Component as an argument (in this case we've called the argument `WrappedComponent`) and returns a new component with the `WrappedComponent` within.

When we return the `Wrapped Component` we have the possibility to manipulate props and to abstract state, even passing state as a prop into the `Wrapped Component`.

We can create `props` passed to the component using `props proxy` pattern as below

```
const propsProxyHOC = (WrappedComponent) => {

    return class extends React.Component {
        render() {
            const newProps = {
                user: currentLoggedInUser
            }

            return <WrappedComponent {...this.props} {...newProps} />
        }
    }
}
```

Props Proxy HOCs are useful to the following situations:

- Manipulating props
- Accessing the instance via Refs (be careful, avoid using refs)
- Abstracting State
- Wrapping/Composing the `WrappedComponent` with other elements

Q100. Explain Inheritance Inversion (*iiHOC*) in react?

Inverted Inheritance HOCs are elementarily expressed like this

```
const inheritanceInversionHOC = (WrappedComponent) => {
    return class extends WrappedComponent {
        render() {
            return super.render()
        }
    }
}
```

Here, the returned class **extends** the `WrappedComponent`. It is called Inheritance Inversion, because instead of the `WrappedComponent` extending some Enhancer class, it is passively extended. In this way the relationship between them seems **inverse**.

Inheritance Inversion gives the HOC access to the `WrappedComponent` instance via this, which means we can use the `state`, `props`, component lifecycle and even the `render` method.

Inversion Inheritance HOCs are useful for the following situations

- Render Highjacking
- Manipulating state

Example:

```
class Welcome extends React.Component {
    render() {
        return (
            <div> Welcome {his.props.user}</div>
        )
    }
}
```

```

const withUser = (WrappedComponent) => {
  return class extends React.Component {
    render() {
      if(this.props.user) {
        return (
          <WrappedComponent {...this.props} />
        )
      }
      return <div>Welcome Guest!</div>
    }
  }
}

const withLoader = (WrappedComponent) => {
  return class extends WrappedComponent {
    render() {
      const { isLoader } = this.props
      if(!isLoaded) {
        return <div>Loading...</div>
      }
      return super.render()
    }
  }
}

export default withLoader(withUser(Welcome))

```

Q101. How to set a dynamic key for state?

Dynamic Key

```

onChange(e) {
  const key = e.target.name
  const value = e.target.value
  this.setState({ [key]: value })
}

```

Nested States

```

handleSetState(cat, key, val) {
  const category = {...this.state[cat]}
  category[key] = val
  this.setState({ [cat]: category })
}

```

Q102. What are the pointer events in React?

Pointer events, in essence, are very similar to mouse events (mousedown, mouseup, etc.) but are hardware-agnostic and thus can handle all input devices such as a mouse, stylus or touch. This is great since it removes the need for separate implementations for each device and makes authoring for cross-device pointers easier.

The API of pointer events works in the same manner as existing various event handlers. Pointer events are added as attributes to React component and are passed a callback that accepts an event. Inside the callback we process the event.

The following event types are now available in React DOM

- onPointerDown
- onPointerMove
- onPointerUp
- onPointerCancel
- onGotPointerCapture
- onLostPointerCapture
- onPointerEnter
- onPointerLeave
- onPointerOver
- onPointerOut

Example: Drag and Drop using Point Events

```
// App Component
import React, { Component } from 'react'
import logo from './logo.svg'
import './App.css'
import DragItem from './DragItem'

class App extends Component {
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <h1 className="App-title">Welcome to React sample of Point
Events</h1>
        </header>
        <div className="App-intro">
          <DragItem />
        </div>
      </div>
    )
  }
}

export default App

DragItem Component
import React from 'react'
const CIRCLE_DIAMETER = 100

export default class DragItem extends React.Component {

  state = {
    gotCapture: false,
    circleLeft: 500,
    circleTop: 100
  }
  isDragging = false
  previousLeft = 0
  previousTop = 0

  onDown = e => {
    this.isDragging = true
    e.target.setPointerCapture(e.pointerId)
```

```
        this.getDelta(e)
    }
    onMouseMove = e => {
        if (!this.isDragging) {
            return
        }

        const {left, top} = this.getDelta(e)
        this.setState(({circleLeft, circleTop}) => ({
            circleLeft: circleLeft + left,
            circleTop: circleTop + top
        }))
    }
    onMouseUp = e => (this.isDragging = false)
    onGotCapture = e => this.setState({gotCapture: true})
    onLostCapture = e => this.setState({gotCapture: false})
    getDelta = e => {
        const left = e.pageX
        const top = e.pageY
        const delta = {
            left: left - this.previousLeft,
            top: top - this.previousTop,
        }
        this.previousLeft = left
        this.previousTop = top

        return delta
    }
    render() {
        const {gotCapture, circleLeft, circleTop} = this.state
        const boxStyle = {
            border: '2px solid #cccccc',
            margin: '10px 0 20px',
            minHeight: 400,
            width: '100%',
            position: 'relative',
        }
        const circleStyle = {
            width: CIRCLE_DIAMETER,
            height: CIRCLE_DIAMETER,
            borderRadius: CIRCLE_DIAMETER / 2,
            position: 'absolute',
            left: circleLeft,
            top: circleTop,
            backgroundColor: gotCapture ? 'red' : 'green',
            touchAction: 'none',
        }
        return (
            <div style={boxStyle}>
                <div
                    style={circleStyle}
                    onPointerDown={this.onDown}
                    onPointerMove={this.onMove}
                    onPointerUp={this.onUp}
                    onPointerCancel={this.onUp}
```

```

        onGotPointerCapture={this.onGotCapture}
        onLostPointerCapture={this.onLostCapture}
    />
</div>
)
}
}

```

Note: It work only in browsers that support the Pointer Events specification

Q103. What is difference between Pure Component vs Component?

PureComponent is exactly the same as Component except that it handles the `shouldComponentUpdate()` method. The major difference between React.PureComponent and React.Component is PureComponent does a shallow comparison on state change. It means that when comparing scalar values it compares their values, but when comparing objects it compares only references. It helps to improve the performance of the app.

A component rerenders every time its parent rerenders, regardless of whether the component's props and state have changed. On the other hand, a pure component will not rerender if its parent rerenders, unless the pure component's props (or state) have changed.

When to use React.PureComponent

- State/Props should be an immutable object
- State/Props should not have a hierarchy
- We should call `forceUpdate` when data changes

Example

```

// Regular class component
class App extends React.Component {
  render() {
    return <h1>Component Example !</h1>
  }
}

// React Pure class component
class Message extends React.Component {
  render() {
    return <h1>PureComponent Example !</h1>
  }
}

```

Q104. How to programmatically redirect to another page using React router?

1. Using `useHistory()`

```

import { useHistory } from "react-router-dom"

function HomeButton() {
  const history = useHistory()

  function handleClick() {
    history.push('/home')
  }
}

```

```

        }

      return (
        <button type="button" onClick={handleClick}>
          Navigate to Home Page
        </button>
      )
    }
  }

```

2. Using withRouter()

```

import { withRouter } from 'react-router-dom'

const Button = withRouter(({ history }) => (
  <button type='button' onClick={() => { history.push('/home') }}>
    Navigate to Home Page
  </button>
))

```

Q105. What is the use of {...this.props} ?

It is called spread operator (ES6 feature) and its aim is to make the passing of props easier.

Example:

```

<div {...this.props}>
  Content Here
</div>

```

It is equal to Class Component

```

const person = {
  name: "Alex",
  age: 26,
  country: "India"
}

```

```

class SpreadExample extends React.Component {
  render() {
    const {name, age, country} = {...this.props}
    return (
      <div>
        <h3> Person Information: </h3>
        <ul>
          <li>name={name}</li>
          <li>age={age}</li>
          <li>country={country}</li>
        </ul>
      </div>
    )
  }
}

ReactDOM.render(
  <SpreadExample {...person}/>
  , mountNode
)

```

Q106. How to pass props in React router?

A component with a render prop takes a function that returns a React element and calls it instead of implementing its own render logic. The **render prop** refers to a technique for sharing code between React components using a prop whose value is a function.

Example:

```
import React from "react"
import { render } from "react-dom"
import { Greeting } from "./components"

import { BrowserRouter as Router, Route, Link } from "react-router-dom"

const styles = {
  fontFamily: "sans-serif",
  textAlign: "center"
}

const App = () => (
  <div style={styles}>
    <h2>Click below to go to other page. Also, open source code</h2>
    <Link to="/greeting/World">Go to /greeting/World</Link>
  </div>
)

const RouterExample = () => (
  <Router>
    <div>
      <ul>
        <li>
          <Link to="/">Home</Link>
        </li>
      </ul>

      <hr />

      <Route exact path="/" component={App} />
      <Route
        path="/greeting/:name"
        render={props => <Greeting text="Hello, " {...props} />}
      />
    </div>
  </Router>
)

render(<RouterExample />, document.getElementById("root"))
import React from "react"

export class Greeting extends React.Component {
  render() {
    const { text, match: { params } } = this.props

    const { name } = params

    return (
      <React.Fragment>
        <h1>Greeting page</h1>
    )
  }
}
```

```

        <p>
          {text} {name}
        </p>
      </React.Fragment>
    )
}
}

```

Q107. How to get query parameters in react routing?

Using useParams()

Example:

```

import React from "react"
import { BrowserRouter as Router, Switch, Route, Link, useParams } from
"react-router-dom"

export default function ParamsExample() {

  return (
    <Router>
      <div>
        <h2>Accounts</h2>
        <ul>
          <li>
            <Link to="/netflix">Netflix</Link>
          </li>
          <li>
            <Link to="/zillow-group">Zillow Group</Link>
          </li>
          <li>
            <Link to="/yahoo">Yahoo</Link>
          </li>
          <li>
            <Link to="/modus-create">Modus Create</Link>
          </li>
        </ul>

        <Switch>
          <Route path="/:id" children={<Child />} />
        </Switch>
      </div>
    </Router>
  )
}

function Child() {
  // We can use the `useParams` hook here to access
  // the dynamic pieces of the URL.
  let { id } = useParams()

  return (
    <div>
      <h3>ID: {id}</h3>
    </div>
  )
}

```

```
)  
}
```

Q108. How do you remove an element in the react state?

Using filter()

In the child component, we need to pass the id of the item we want to delete to the parent.

```
// Item.js  
import React, { Component } from "react"  
  
class Item extends Component {  
  state = {  
    count: this.props.item.value  
  }  
  
  handleIncrement = e => {  
    this.setState({ count: this.state.count + 1 })  
  }  
  
  render() {  
    return (  
      <React.Fragment>  
        <div className="card mb-2">  
          <h5 className={this.styleCardHeader()}>{this.styleCount()}</h5>  
          <div className="card-body">  
            <button  
              onClick={item => {  
                this.handleIncrement({ item })  
              }}  
              className="btn btn-lg btn-outline-secondary"  
            >  
              Increment  
            </button>  
  
            <button  
              onClick={() => this.props.onDelete(this.props.item.id)}  
              className="btn btn-lg btn-outline-danger ml-4"  
            >  
              Delete  
            </button>  
          </div>  
        </div>  
      </React.Fragment>  
    )  
  }  
  
  styleCardHeader() {  
    let classes = "card-header h4 text-white bg-"  
    classes += this.state.count === 0 ? "warning" : "primary"  
    return classes  
  }  
  
  styleCount() {  
    const { count } = this.state
```

```

        return count === 0 ? "No Items!" : count
    }
}

export default Item
Now in the parent component, we need to update the handleDelete() function to accept that id as a parameter. In addition, we need to use the filter function to create a new array of items which does not contain the item which was clicked. Then we have to call the setState() function to update the state.

import React, { Component } from "react"
import Item from "./item"

class Items extends Component {
  state = {
    items: [{ id: 1, value: 0 }, { id: 2, value: 10 }, { id: 3, value: 0 }]
  }

  handleDelete = itemId => {
    const items = this.state.items.filter(item => item.id !== itemId)
    this.setState({ items })
  }

  render() {
    return (
      <React.Fragment>
        {this.state.items.map(item => (
          <Item
            key={item.id}
            onDelete={this.handleDelete}
            item={item}
          />
        )))
      </React.Fragment>
    )
  }
}

export default Items

```

Q109. What is Destructuring in React?

Destructuring is a convenient way of accessing multiple properties stored in objects and arrays. It was introduced to JavaScript by ES6 and has provided developers with an increased amount of utility when accessing data properties in Objects or Arrays.

When used, destructuring does not modify an object or array but rather copies the desired items from those data structures into variables. These new variables can be accessed later on in a React component.

Destructuring in JS

Without destructuring

```
const person = {
  firstName: "Alex",
  lastName: "K",
```

```

    age: 25,
    sex: ""
}

const first = person.firstName
const age = person.age
const sex = person.sex || "Male"

console.log(first) // "Alex"
console.log(age) // 25
console.log(sex) // Male --> default value

```

With destructuring

```

const person = {
  firstName: "Alex",
  lastName: "K",
  age: 25,
  sex: "M"
}

const { firstName, lastName, age, sex } = person

console.log(firstName) // Alex
console.log(lastName) // K
console.log(age) // 25
console.log(sex) // M

```

Destructuring in React

Example:

```

import React from 'react'
import Button from '@material-ui/core/Button'

```

```

export default function Events() {

  const [counter, setcounter] = React.useState(0)

  return (
    <div className='Counter'>
      <div>Result: {counter}</div>
      <Button
        variant='contained'
        color='primary'
        onClick={() => setcounter(counter + 1)}
      >
        Increment
      </Button>

      <Button
        variant='contained'
        color='primary'
        onClick={() => setcounter((counter > 0) ? (counter - 1) : 0)}
      >
        Decrement
      </Button>
    </div>
  )
}
```

```
)  
}
```

Q110. What is the difference between NavLink and Link?

The `<Link>` component is used to navigate the different routes on the site. But `<NavLink>` is used to add the style attributes to the active routes.

Link

```
<Link to="/">Home</Link>
```

NavLink

```
<NavLink to="/" activeClassName="active">Home</NavLink>
```

Example:

index.css

```
.active {  
    color: blue;  
}
```

Routes.js

```
import ReactDOM from 'react-dom'  
import './index.css'  
import { Route, NavLink, BrowserRouter as Router, Switch } from 'react-router-dom'  
import App from './App'  
import Users from './users'  
import Contact from './contact'  
import Notfound from './notfound'  
  
const Routes = (  
    <Router>  
        <div>  
            <ul>  
                <li>  
                    <NavLink exact activeClassName="active" to="/">  
                        Home  
                    </NavLink>  
                </li>  
                <li>  
                    <NavLink activeClassName="active" to="/users">  
                        Users  
                    </NavLink>  
                </li>  
                <li>  
                    <NavLink activeClassName="active" to="/contact">  
                        Contact  
                    </NavLink>  
                </li>  
            </ul>  
            <hr />  
            <Switch>  
                <Route exact path="/" component={App} />  
                <Route path="/users" component={Users} />  
                <Route path="/contact" component={Contact} />  
                <Route component={Notfound} />  
            </Switch>  
    </Router>  
)
```

```

        </div>
    </Router>
}

ReactDOM.render(Routes, document.getElementById('root'))

```

Q111. What is `withRouter` for in react-router-dom?

`withRouter()` is a higher-order component that allows to get access to the `history` object's properties and the closest `<Route>`'s match. `withRouter` will pass updated `match`, `location`, and `history` props to the wrapped component whenever it renders.

Example:

```

import React from "react"
import PropTypes from "prop-types"
import { withRouter } from "react-router"

// A simple component that shows the pathname of the current location
class ShowTheLocation extends React.Component {
    static propTypes = {
        match: PropTypes.object.isRequired,
        location: PropTypes.object.isRequired,
        history: PropTypes.object.isRequired
    }

    render() {
        const { match, location, history } = this.props

        return <div>You are now at {location.pathname}</div>
    }
}

const ShowTheLocationWithRouter = withRouter(ShowTheLocation)

```

Q112. How to display API data using Axios in React?

Axios is a promise based HTTP client for making HTTP requests from a browser to any web server.

Features

- **Interceptors:** Access the request or response configuration (headers, data, etc) as they are outgoing or incoming. These functions can act as gateways to check configuration or add data.
- **Instances:** Create reusable instances with `baseURL`, `headers`, and other configuration already set up.
- **Defaults:** Set default values for common headers (like `Authorization`) on outgoing requests. This can be useful if you are authenticating to a server on every request.

Installation

```
npm install axios -- save
```

Shorthand Methods

- `axios.request(config)`
- `axios.get(url[, config])`
- `axios.delete(url[, config])`

- `axios.head(url[, config])`
- `axios.options(url[, config])`
- `axios.post(url[, data[, config]])`
- `axios.put(url[, data[, config]])`
- `axios.patch(url[, data[, config]])`

POST Request Example

```
axios.post('/url', {data: 'data'})
  .then((res)=>{
    //on success
  })
  .catch((error)=>{
    //on error
  })
```

GET Request Example

```
axios.get('/url')
  .then((res)=>{
    //on success
  })
  .catch((error)=>{
    //on error
  })
```

Performing Multiple Concurrent Requests Example

```
function getUserAccount() {
  return axios.get('/user/12345')
}

function getUserPermissions() {
  return axios.get('/user/12345/permissions')
}

axios.all([getUserAccount(), getUserPermissions()])
  .then(axios.spread(function (acct, perms) {
    // Both requests are now complete
  }))
```

Example: Making a POST Request

```
import React from 'react'
import axios from 'axios'

export default class PersonList extends React.Component {
  state = {
    name: '',
  }

  handleChange = event => {
    this.setState({ name: event.target.value })
  }

  handleSubmit = event => {
    event.preventDefault()

    const user = {
      name: this.state.name
    }
```

```

        axios.post(`https://jsonplaceholder.typicode.com/users`, { user })
          .then(res => {
            console.log(res)
            console.log(res.data)
          })
      }

      render() {
        return (
          <div>
            <form onSubmit={this.handleSubmit}>
              <label>
                Person Name:
                <input type="text" name="name" onChange={this.handleChange} />
              </label>
              <button type="submit">Add</button>
            </form>
          </div>
        )
      }
    }
}

```

Q113. How to translate your React app with react-i18next?

Installing dependencies

```
npm install react-i18next i18next --save
```

Configure i18next

Create a new file `i18n.js` beside your `index.js` containing following content:

```

import i18n from "i18next"
import { initReactI18next } from "react-i18next"

// Translations
const resources = {
  en: {
    translation: {
      "welcome.title": "Welcome to React and react-i18next"
    }
  }
}

i18n
  .use(initReactI18next) // passes i18n down to react-i18next
  .init({
    resources,
    lng: "en",
    keySeparator: false, // we do not use keys in form messages.welcome
    interpolation: {
      escapeValue: false // react already safes from xss
    }
  })

export default i18n

```

we pass the i18n instance to react-i18next which will make it available for all the components via the context api.

```
import React, { Component } from "react"
import ReactDOM from "react-dom"
import './i18n'
import App from './App'

// append app to dom
ReactDOM.render(
  <App />,
  document.getElementById("root")
)
```

Using the Hook

The t function is the main function in i18next to translate content.

```
import React from 'react'
import { useTranslation } from 'react-i18next'

function MyComponent () {
  const { t, i18n } = useTranslation()
  return <h1>{t('welcome.title')}</h1>
}
```

Using the HOC

Using higher order components is one of the most used method to extend existing components by passing additional props to them. The t function is in i18next the main function to translate content.

```
import React from 'react'
import { withTranslation } from 'react-i18next'

class HighOrderComponent extends React.Component {
  render() {

    return (
      <h1>{this.props.t('welcome.title')}</h1>
    )
  }
}

export default withTranslation()(HighOrderComponent)
[.]
```

Q114. How RxJS is used in React for state management?

RxJS is a library for reactive programming using Observables, to make it easier to compose asynchronous or callback-based code. Reactive programming is an event-based paradigm that allows us to run asynchronous sequences of events as soon as data is pushed to a consumer.

RxJS Terminology

- **Observable:** An Observable is a data stream that houses data that can be passed through different threads.
- **Observer:** An Observer consumes the data supplied by an Observable
- **Subscription:** In order for Observer to consume data from Observable, Observer has to subscribe it to the Observable.

- **Subject:** An RxJS Subject can act as both an Observable and an Observer at the same time. In this way, values can be multicasted to many Observers from it so that when a Subject receives any data, that data can be forwarded to every Observer subscribed to it.
- **Operators:** Operators are methods that can be used on Observables and subjects to manipulate, filter or change the Observable in a specified manner into a new Observable.
- **BehaviorSubject:** It allows multiple observers to listen on stream and events multicasted to the observers, BehaviorSubject stores the latest value and broadcasts it to any new subscribers.

Example:

```
// messageService.js
import { BehaviourSubject } from 'rxjs'

const subscriber = new BehaviourSubject(0)

const messageService = {
  send: function(msg) {
    subscriber.next(msg)
  }
}
export {
  messageService,
  subscriber
}
```

The messageService object has a send function, which takes a msg parameter which holds the data we need to broadcast all listening components, in the function body we call the emit method in the subscriber object it multicasts the data to the subscribing components.

```
import React, { Component } from 'react'
import { render } from 'react-dom'
import './style.css'
import { subscriber, messageService } from './messageService'

class ConsumerA extends React.Component {
  constructor() {
    this.state = {
      counter: 0
    }
  }

  componentDidMount() {
    subscriber.subscribe((v) => {
      let { counter } = this.state
      counter = counter + v
      this.setState({ counter })
    })
  }

  render() {
    let { counter } = this.state
    return (
      <div>
        <hr/>
        <h3> Counter for Consumer A </h3>
    
```

```

        <div> Counter: {counter} </div>
        <hr/>
    </div>
)
}
}

class ConsumerB extends React.Component {
constructor() {
    this.state = {
        counter: 0
    }
}
componentDidMount() {
    subscriber.subscribe((v) => {
        let { counter } = this.state
        counter = counter + v
        this.setState({ counter })
    })
}

render() {
    let { counter } = this.state
    return (
        <div>
            <hr/>
            <h3>Counter for Consumer B</h3>
            <div> Counter: { counter } </div>
            <hr/>
            <ProducerB />
        </div>
    )
}
}

class ProducerA extends React.Component {
render() {
    return (
        <div>
            <h3>ProducerA</h3>
            <button onClick={ (e) => subscriber.next(1) }>Increment Counter</button>
            <ConsumerA />
        </div>
    )
}
}

class ProducerB extends React.Component {
render() {
    return (
        <div>
            <h3>ProducerB</h3>
            <button onClick={ (e) => subscriber.next(-1) }>Decrement
Counter</button>
        </div>
    )
}
}
```

```

        )
    }
}

class App extends Component {
  render() {
    return (
      <div>
        <ProducerA />
        <hr/>
        <ConsumerB />
      </div>
    )
  }
}

render(<App/>, document.getElementById('root'));

```

The ConsumerA and ConsumerB components keep a state counter individual. In their componentDidMount they subscribe to the same stream subscriber, anytime an event is published they both update the counter. The ProducerA and ProducerB have buttons Increment Counter and Decrement Counter when clicked they emit 1 or -1.

[_]

Q115. What is lazy function in React?

`React.lazy()` makes it easy to create components that are loaded using dynamic `import()` but are rendered like regular components. This will automatically cause the bundle containing the component to be loaded when the component is rendered.

`React.lazy()` takes a function as its argument that must return a promise by calling `import()` to load the component. The returned Promise resolves to a module with a default export containing the React component.

Example:

```

import React, { lazy } from 'react'

const MyComponent = React.lazy(() => import('./MyComponent'))

const App = () => {
  <div>
    <MyComponent />
  </div>
}

```

Q116. What are the benefits of using Axios() over Fetch() for making http requests?

Fetch: The Fetch API provides a `fetch()` method defined on the window object. It also provides a JavaScript interface for accessing and manipulating parts of the HTTP pipeline (requests and responses). The `fetch` method has one mandatory argument- the URL of the resource to be fetched. This method returns a Promise that can be used to retrieve the response of the request.

Example:

```

fetch('path-to-the-resource-to-be-fetched')
  .then((response) => {
    // Code for handling the response
  })
  .catch((error) => {
    // Error Handling
  });

```

Axios: Axios is a Javascript library used to make HTTP requests from node.js or XMLHttpRequests from the browser and it supports the Promise API that is native to JS ES6. It can be used intercept HTTP requests and responses and enables client-side protection against XSRF. It also has the ability to cancel requests.

Example:

```

axios.get('url')
  .then((response) => {
    // Code for handling the response
  })
  .catch((error) => {
    // Error Handling
  });

```

Differences between Axios and Fetch

| Axios() | Fetch() |
|--|--|
| Axios has url in request object. | Fetch has no url in request object. |
| Axios is a stand-alone third party package that can be easily installed. | Fetch is built into most modern browsers |
| Axios has built-in XSRF protection. | Fetch does not. |
| Axios uses the data property. | Fetch uses the body property. |
| Axios data contains the object. | Fetch's body has to be stringified. |
| Axios request is ok when status is 200 and statusText is 'OK'. | Fetch request is ok when response object contains the ok property. |
| Axios performs automatic transforms of JSON data. | Fetch is a two-step process when handling JSON data- first, to make the actual request; second, to call the .json() method on the response. |
| Axios allows cancelling request and request timeout. | Fetch does not. |
| Axios has the ability to intercept HTTP requests. | Fetch, by default, doesn't provide a way to intercept requests. |
| Axios has built-in support for download progress. | Fetch does not support upload progress. |
| Axios has wide browser support. | Fetch only supports Chrome 42+, Firefox 39+, Edge 14+, and Safari 10.1+. |

Q117. What is the difference between rendering and mounting in ReactJS?

Rendering is any time a function component gets called (or a class-based render method gets called) which returns a set of instructions for creating DOM. `render()` function will be invoked every time rerendering happens in the component. It may happen either through a state change or a prop change.

Mounting is when React `renders` the component for the first time and actually builds the initial DOM from those instructions. Mounting a react component means the actual addition of the DOM elements created by the react component into the browser DOM for the first time.

A **re-render** is when React calls the function component again to get a new set of instructions on an already mounted component.

Example:

```
class App extends React.Component {  
  state = {  
    showUser: false  
  }  
  
  render() {  
    return (  
      <div>  
        {this.state.showUser && <User name="Brad" />}  
        <button onClick={() => this.setState({ showUser: true })}>  
          Show User  
        </button>  
        <button onClick={() => this.setState({ showUser: false })}>  
          Hide User  
        </button>  
      </div>  
    )  
  }  
}
```

```
ReactDOM.render(<App />, document.getElementById('root'))
```

Internally, React will create an instance of `App` and will eventually call the `render()` method to get the first set of instructions for what it needs to build in the DOM. Anytime React calls the `render` method of a class-based component, we call that a **render**.

Q118. What is Flow in react?

Type Checking

Type checking means ensuring that the type of a property (variable, object, function, string) in a programming language is being used as it should be. It is the process of verifying and enforcing the constraints of types, and it can occur either at compile time or at runtime. It helps to detect and report errors.

Type checking can be divided into two: static type checking and dynamic type checking.

1. Static Type Checking

Static type checking is used in static-typed languages where the type of the variable is known at the compile time. This means that the type of the variable must be declared beforehand. Static

typing usually results in compiled code that executes more quickly because the compiler already knows the exact data types that are in use.

2. Dynamic type checking

Dynamic type checking is used in dynamic-typed languages where the type is usually known at runtime. This means that the type of the variable doesn't need to be explicitly defined.

Flow

Flow is a static type checker for JavaScript apps that aims to find and eliminate problems as you code. Designed by the Facebook team for JavaScript developers, it's a static type checker that catches common errors in your application before they run.

Integrating Flow

```
# Create React App with Flowchecker
npx create-react-app flowchecker
```

```
# Add Dependency
npm install --save-dev flow-bin
```

The next thing to do is add Flow to the "scripts" section of your package.json so that Flow can be used in the terminal. In the package.json file, add the code snippet below.

```
"scripts": {
  "flow": "flow",
}
```

Finally, for the Flow setup, run any of the commands below:

```
npm run flow init
```

This will help to create a Flow configuration file that should be committed. The Flow config file helps to determine the files that Flow should work with and what should be ignored.

Q119. What is an alternative way to avoid having to bind to this in event callback methods?

Bind in Constructor

```
class App extends Component {

  constructor(props) {
    super(props)
    this.handleClick = this.handleClick.bind(this)
  }
  handleClick() {
    console.log('Clicked !')
  }
  render() {
    return <button onClick={this.handleClick}>Click Me</button>
  }
}
```

Bind in Render

```
class App extends Component {

  handleClick() {
    console.log('Clicked !')
  }
  render() {
    return <button onClick={this.handleClick.bind(this)}>Click Me</button>
  }
}
```

```

        }
    }

Arrow Function in Render
class App extends Component {

    handleClick() {
        console.log('Clicked !')
    }
    render() {
        return <button onClick={() => this.handleClick()}>Click Me</button>
    }
}

```

Using an arrow function in render creates a new function each time the component renders, which may break optimizations based on strict identity comparison.

Q120. Why is it advised to pass a callback function to setState as opposed to an object?

Because `this.props` and `this.state` may be updated asynchronously, we should not rely on their values for calculating the next state.

Example: setState Callback in a Class Component

```

import React, { Component } from 'react'

class App extends Component {
    constructor(props) {
        super(props)
        this.state = {
            age: 0,
        }
    }

    // this.checkAge is passed as the callback to setState
    updateAge = (value) => {
        this.setState({ age: value}, this.checkAge)
    }

    checkAge = () => {
        const { age } = this.state
        if (age !== 0 && age >= 21) {
            // Make API call to /beer
        } else {
            // Throw error 404, beer not found
        }
    }

    render() {
        const { age } = this.state
        return (
            <div>
                <p>Drinking Age Checker</p>
                <input
                    type="number"
                    value={age}

```

```

        onChange={e => this.updateAge(e.target.value)}
      />
    </div>
  )
}
}

export default App

```

Example: setState Callback in a Functional Component

```

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

function App() {
  const [age, setAge] = useState(0)

  updateAge(value) {
    setAge(value)
  }

  useEffect(() => {
    if (age !== 0 && age >= 21) {
      // Make API call to /beer
    } else {
      // Throw error 404, beer not found
    }
  }, [age])
}

return (
  <div>
    <p>Drinking Age Checker</p>
    <input
      type="number"
      value={age}
      onChange={e => setAge(e.target.value)} />
  </div>
)
}

export default App

```

Q121. What is the alternative of binding `this` in the constructor?

Arrow Function: This creates and binds the function all at once. Inside render (and elsewhere), the function is already bound because the arrow function preserves the this binding.

Example:

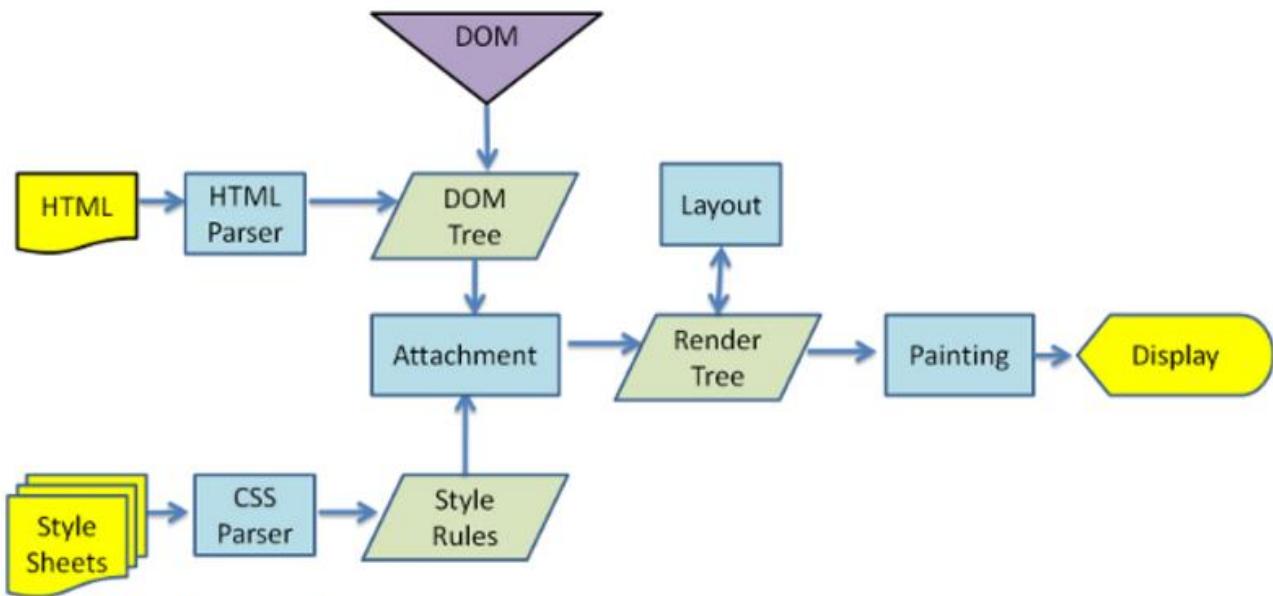
```

class Button extends React.Component {
  // no binding
  handleClick = (e) => {
    console.log('clicked !');
  }
  render() {
    return <button onClick={this.handleClick}>Click Me</button>;
  }
}

```

}

Q122. What is the difference between ShadowDOM and VirtualDOM?



Document Object Model

It is a way of representing a structured document via objects. It is cross-platform and language-independent convention for representing and interacting with data in HTML, XML, and others. Web browsers handle the DOM implementation details, so we can interact with it using JavaScript and CSS.

Virtual DOM

Virtual DOM is any kind of representation of a real DOM. Virtual DOM is about avoiding unnecessary changes to the DOM, which are expensive performance-wise, because changes to the DOM usually cause re-rendering of the page. It allows to collect several changes to be applied at once, so not every single change causes a re-render, but instead re-rendering only happens once after a set of changes was applied to the DOM.

Shadow DOM

Shadow DOM is mostly about encapsulation of the implementation. A single custom element can implement more-or-less complex logic combined with more-or-less complex DOM. Shadow DOM refers to the ability of the browser to include a subtree of DOM elements into the rendering of a document, but not into the main document DOM tree.

Difference

The virtual DOM creates an additional DOM. The shadow DOM simply hides implementation details and provides isolated scope for web components.

Q123. What is Lifting State Up in ReactJS?

The common approach to share state between two components is to move the state to common parent of the two components. This approach is called as lifting state up in React.js. With the shared state, changes in state reflect in relevant components simultaneously.

Example:

The App component containing PlayerContent and PlayerDetails component. PlayerContent shows the player name buttons. PlayerDetails shows the details of the in one line.

The app component contains the state for both the component. The selected player is shown once we click on the one of the player button.

App.js

```
import React from 'react'
import PlayerContent from './PlayerContent'
import PlayerDetails from './PlayerDetails'
import './App.css'

class App extends React.Component {
  constructor(props) {
    super(props)
    this.state = { selectedPlayer: [0,0], playerName: '' }
    this.updateSelectedPlayer = this.updateSelectedPlayer.bind(this)
  }
  updateSelectedPlayer(id, name) {
    var arr = [0, 0, 0, 0]
    arr[id] = 1
    this.setState({
      playerName: name,
      selectedPlayer: arr
    })
  }
  render () {
    return (
      <div>
        <PlayerContent active={this.state.selectedPlayer[0]} clickHandler={this.updateSelectedPlayer} id={0} name="David"/>
        <PlayerContent active={this.state.selectedPlayer[1]} clickHandler={this.updateSelectedPlayer} id={1} name="Steve"/>
        <PlayerDetails name={this.state.playerName}/>
      </div>
    )
  }
}
export default App
```

PlayerContent.js

```
import React , { Component } from 'react'

class PlayerContent extends Component {
  render () {
    return (
      <button
        onClick={() => {this.props.clickHandler(this.props.id, this.props.name)}}
        style={{color: this.props.active? 'red': 'blue'}}
      >
        {this.props.name}
      </button>
    )
  }
}
```

```

}

export default PlayerContent
PlayerDetails.js
import React, { Component } from 'react'

class PlayerDetails extends Component {
  render () {
    return (
      <div>{this.props.name}</div>
    )
  }
}

export default PlayerDetails

```

Q124. What is "Children" in React?

The children, in React, refer to the generic box whose contents are unknown until they're passed from the parent component. Children allows to pass components as data to other components, just like any other prop you use. The special thing about children is that React provides support through its `ReactElement API` and `JSX`. XML children translate perfectly to React children!

Example:

```

const Picture = (props) => {
  return (
    <div>
      <img src={props.src}/>
      {props.children}
    </div>
  )
}

```

This component contains an `` that is receiving some props and then it is displaying `{props.children}`. Whenever this component is invoked `{props.children}` will also be displayed and this is just a reference to what is between the opening and closing tags of the component.

//App.js

```

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

```

Q125. What does Eject do in Create React App?

The `create-react-app` commands generate **React App** with an excellent configuration and helps you build your React app with the best practices in mind to optimize it. However, running the `eject` script will remove the single build dependency from your project. That means it will copy the configuration files and the transitive dependencies (e.g. `Webpack`, `Babel`, etc.) as

dependencies in the package.json file. If you do that, you'll have to ensure that the dependencies are installed before building your project.

After running the eject, commands like npm start and npm run build will still work, but they will point to the copied scripts so you can tweak them. It won't be possible to run it again since all scripts will be available except the eject one.

Q126. Why are string refs considered legacy in React?

Reactjs provides a way to get references to dom elements that react is rendering through jsx.

Previously, it was through what are now legacy refs:

```
componentWillUpdate() {  
  this.refs.example.tagName == "div";  
}  
  
render() {  
  return (  
    <div ref="example"/>  
  )  
}
```

Where we can assign an element an identifier and react would keep a `refs` hash up to date with references to the dom for that element.

The new react version uses callbacks

```
render() {  
  return (  
    <div ref={(div) => { console.log('tag name:', div.tagName); }} />  
  )  
}
```

This callback is called when the component mounts with a reference to the dom element as an argument. Importantly, when the component unmounts the callback is called again but this time with `null` as an argument.

Q127. What are the recommended way for static type checking?

Static type checkers like Flow and TypeScript identify certain types of problems before you even run your code. They can also improve developer workflow by adding features like auto-completion. For this reason, we should use Flow or TypeScript instead of PropTypes for larger code bases.

Q128. What is the difference between Flow and PropTypes?

Flow is a static analysis tool which uses a superset of the language, allows to add type annotations to all of your code and catch an entire class of bugs at compile time.

PropTypes is a basic type checker which has been patched onto React. It can't check anything other than the types of the props being passed to a given component.

Q129. What is Compound Components in React?

A compound component is a type of component that manages the internal state of a feature while delegating control of the rendering to the place of implementation opposed to the point of declaration. They provide a way to shield feature specific logic from the rest of the app providing a clean and expressive API for consuming the component.

Internally they are built to operate on a set of data that is passed in through children instead of props. Behind the scenes they make use of React's lower level API such as

`React.children.map()`, and `React.cloneElement()`. Using these methods, the component is able to express itself in such a way that promotes patterns of composition and extensibility.

Example:

```
function App() {
  return (
    <Menu>
      <MenuButton>
        Actions <span aria-hidden>▼</span>
      </MenuButton>
      <MenuList>
        <MenuItem onSelect={() => alert('Download')}>Download</MenuItem>
        <MenuItem onSelect={() => alert('Copy')}>Create a Copy</MenuItem>
        <MenuItem onSelect={() => alert('Delete')}>Delete</MenuItem>
      </MenuList>
    </Menu>
  )
}
```

In this example, the `<Menu>` establishes some shared implicit state. The `<MenuButton>`, `<MenuList>`, and `<MenuItem>` components each access and/or manipulate that state, and it's all done implicitly. This allows you to have the expressive API you're looking for.

Q130. What is React Fiber?

React Fiber is the new **reconciliation algorithm**. Reconciliation is the process of comparing or diffing old trees with a new tree in order to find what is changed or modified. In the original reconciliation algorithm (now called **Stack Reconciler**), the processing of component trees was done synchronously in a single pass, so the main thread was not available for other UI related tasks like animation, layouts, and gesture handling. Fiber Reconciler has different goals:

- Ability to split interruptible work in chunks.
- Ability to prioritize, rebase, and reuse work in progress.
- Ability to yield back and forth between parents and children to support layout in React.
- Ability to return multiple elements from `render()`.

A fiber is a JavaScript object that contains information about a component, its input, and output. At any time, a component instance has at most two fibers that correspond to it: the current fiber and the work-in-progress fiber. A fiber can be defined as a unit of work.

React Fiber performs reconciliation in two phases: Render and Commit

1. Lifecycle methods called during render phase:

- `UNSAFE_componentWillMount()`
- `UNSAFE_componentWillReceiveProps()`
- `getDerivedStateFromProps()`
- `shouldComponentUpdate()`
- `UNSAFE_componentWillUpdate()`

- render()

2. Lifecycle methods called during commit phase:

- getSnapshotBeforeUpdate()
- componentDidMount()
- componentDidUpdate()
- componentWillUnmount()

The earlier whole reconciliation process was synchronous (recursive), but in Fiber, it is divided into two phases. Render phase (a.k.a. Reconciliation phase) is asynchronous, so three of the lifecycle methods were marked unsafe because putting the code with side-effects inside these methods can cause problems, as lifecycle methods of different components are not guaranteed to fire in a predictable order.

React Fiber uses `requestIdleCallback()` to schedule the low priority work and `requestAnimationFrame()` to schedule high priority work.

Problems with Current Implementation:

- Long-running tasks cause frame drops.
- Different tasks have different priorities.

How React Fiber works

- It makes apps more fluid and responsible.
- In the future, it could parallelize work a.k.a. Time Slicing.
- It would improve startup time while rendering components using React Suspense.

Fiber is currently available for use but it runs in compatibility mode with the current implementation.

Q131. Explain Composition vs Inheritance in React?

Inheritance

Inheritance is a concept in object-oriented programming in which one class inherits properties and methods of another class. This is useful in code reusability.

Example:

```
class UserNameForm extends React.Component {
  render() {
    return (
      <div>
        <input type="text" />
      </div>
    )
  }
}

class CreateUserName extends UserNameForm {
  render() {
    const parent = super.render();
    return (
      <div>
        {parent}
        <button>Create</button>
      </div>
    )
  }
}

class UpdateUserName extends UserNameForm {
```

```

        render() {
            const parent = super.render();
            return (
                <div>
                    {parent}
                    <button>Update</button>
                </div>
            )
        }
    }
ReactDOM.render(
    (<div>
        < CreateUserName />
        < UpdateUserName />
    </div>), document.getElementById('root')
)

```

Here, We extended the `UserNameForm` component and extracted its method in child component using `super.render()`

Composition

Composition is also a familiar concept in Object Oriented Programming. Instead of inheriting properties from a base class, it describes a class that can reference one or more objects of another class as instances.

Example:

```

class UserNameForm extends React.Component {
    render() {
        return (
            <div>
                <input type="text" />
            </div>
        );
    }
}

class CreateUserName extends React.Component {
    render() {
        return (
            <div>
                < UserNameForm />
                <button>Create</button>
            </div>
        )
    }
}

class UpdateUserName extends React.Component {
    render() {
        return (
            <div>
                < UserNameForm />
                <button>Update</button>
            </div>
        )
    }
}

ReactDOM.render(

```

```

        (<div>
          <CreateUserName />
          <UpdateUserName />
        </div>), document.getElementById('root')
)

```

Inheritance vs Composition

Inheritance used the `is-a` relationship method. Derived components had to inherit the properties of the base component and it was quite complicated while modifying the behavior of any component.

Composition does not inherit properties, only the behavior. In inheritance, it was difficult to add new behavior because the derived component was inheriting all the properties of parent class and it was quite difficult to add new behavior. But in composition, we only inherit behavior and adding new behavior is fairly simple and easy.

React recommends use of Composition over Inheritance, here is why. Everything in React is a component, and it follows a strong component based model. This is one of the primary reasons that composition is a better approach than inheritance for code reuse.

Q132. What is a Webhook in React?

Web hooks provide a mechanism where by a server-side application can notify a client-side application when a new event (that the client-side application might be interested in) has occurred on the server.

Webhooks are also sometimes referred to as "Reverse APIs". In APIs, the client-side application calls (consumes) the server-side application. Whereas, in case of web hooks it is the server-side that calls (consumes) the web hook (the end-point URL provided by the client-side application), i.e. it is the server-side application that calls the client-side application.

Example: ToDo

Q133. Explain is `useCallback()`, `useMemo()`, `useImperativeHandle()`, `useLayoutEffect()`, `useDebugValue()` in React?

1. `useCallback()`

React's `useCallback()` Hook can be used to optimize the rendering behavior of your React function components. The `useCallback` will return a memoized version of the callback that only changes if one of the dependencies has changed. This is useful when passing callbacks to optimized child components that rely on reference equality to prevent unnecessary renders (e.g. `shouldComponentUpdate`).

```

function App() {

  const memoizedHandleClick = useCallback(
    () => console.log('Click happened'), []
  ) // Tells React to memoize regardless of arguments.

  return <Button onClick={memoizedHandleClick}>Click Me</Button>
}

```

2. `useMemo()`

React's `useMemo()` Hook can be used to optimize the computation costs of your React function components. The `useMemo()` is similar to `useCallback()` except it allows you to apply memoization to any value type (not just functions). It does this by accepting a function which returns the value and then that function is only called when the value needs to be retrieved (which typically will only happen once each time an element in the dependencies array changes between renders).

Example:

React application which renders a list of users and allows us to filter the users by their name. The filter happens only when a user explicitly clicks a button; not already when the user types into the input field.

```
import React from 'react'

const users = [
  { id: 'a', name: 'Robin' },
  { id: 'b', name: 'Dennis' },
]

const App = () => {
  const [text, setText] = React.useState('')
  const [search, setSearch] = React.useState('')

  const handleText = (event) => {
    setText(event.target.value)
  }

  const handleSearch = () => {
    setSearch(text)
  }

  // useMemo Hooks
  const filteredUsers = React.useMemo(
    () =>
      users.filter((user) => {
        console.log('Filter function is running ...');
        return user.name.toLowerCase().includes(search.toLowerCase());
      }),
    [search]
  );

  return (
    <div>
      <input type="text" value={text} onChange={handleText} />
      <button type="button" onClick={handleSearch}>
        Search
      </button>

      <List list={filteredUsers} />
    </div>
  )
}

const List = ({ list }) => {
  return (

```

```

        <ul>
          {list.map((item) => (
            <ListItem key={item.id} item={item} />
          )))
        </ul>
      )
    }

const ListItem = ({ item }) => {
  return <li>{item.name}</li>
}

export default App

```

Here, the **filteredUsers** function is only executed once the search state changes. It doesn't run if the text state changes, because that's not a dependency for this filter function and thus not a dependency in the dependency array for the `useMemo` hook.

3. `useImperativeHandle()`

`useImperativeHandle()` customizes the instance value that is exposed to parent components when using `ref`. As always, imperative code using `refs` should be avoided in most cases.

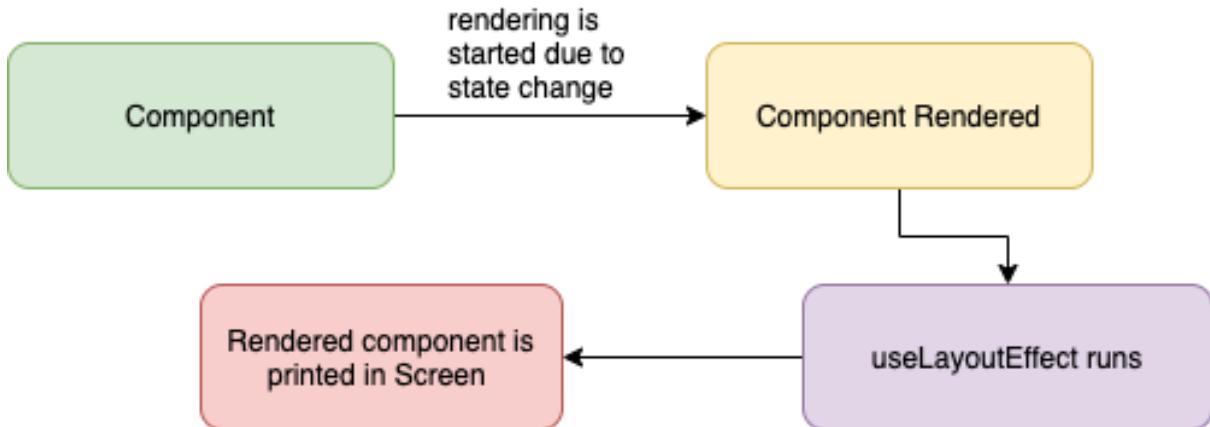
`useImperativeHandle` should be used with `forwardRef`.

```

function FancyInput(props, ref) {
  const inputRef = useRef()
  useImperativeHandle(ref, () => ({
    focus: () => {
      inputRef.current.focus()
    }
  }))
  return <input ref={inputRef} ... />
}
FancyInput = forwardRef(FancyInput)

```

4. `useLayoutEffect()`



This runs synchronously immediately after React has performed all DOM mutations. 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.

As far as scheduling, this works the same way as `componentDidMount` and `componentDidUpdate`. Your code runs immediately after the DOM has been updated, but before the browser has had a chance to "paint" those changes (the user doesn't actually see the updates until after the browser has repainted).

Example:

```
import React, { useState, useLayoutEffect } from 'react'
import ReactDOM from 'react-dom'

const BlinkyRender = () => {
  const [value, setValue] = useState(0)

  useLayoutEffect(() => {
    if (value === 0) {
      setValue(10 + Math.random() * 200)
    }
  }, [value])

  console.log('render', value)

  return (
    <div onClick={() => setValue(0)}>
      value: {value}
    </div>
  )
}

ReactDOM.render( <BlinkyRender />, document.querySelector('#root'))
```

useLayoutEffect vs useEffect

- **useLayoutEffect:** If you need to mutate the DOM and/or do need to perform measurements
- **useEffect:** If you don't need to interact with the DOM at all or your DOM changes are unobservable (seriously, most of the time you should use this).

5. useDebugValue()

useDebugValue() can be used to display a label for custom hooks in React DevTools.

Example:

```
function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null)

  // ...

  // Show a label in DevTools next to this Hook
  // e.g. "FriendStatus: Online"
  useDebugValue(isOnline ? 'Online' : 'Offline')

  return isOnline
}
```

Q134. How does Axios Interceptors work in react?

Axios interceptors are functions that Axios calls for every request. We can use interceptors to transform the request before Axios sends it, or transform the response before Axios returns the response to our code.

There are two types of interceptors:

- **request interceptor:** This is called before the actual call to the endpoint is made.

- **response interceptor:** This is called before the promise is completed and the data is received by the then callback.

1. Request interceptor

One common use case for a request handler is to modify or add new HTTP headers. For example, an authentication token could be injected into all requests.

Example:

```
// Add request handler
const requestHandler = (request) => {
  if (isHandlerEnabled(request)) {
    // Modify request here
    request.headers['X-Auth'] = 'https://example.com/vPvKWe'
  }
  return request
}

// Enable request interceptor
axiosInstance.interceptors.request.use(
  request => requestHandler(request)
)
```

2. Response and error interceptors

Example:

```
// Add response handlers
const errorHandler = (error) => {
  if (isHandlerEnabled(error.config)) {
    // Handle errors
  }
  return Promise.reject({ ...error })
}

const successHandler = (response) => {
  if (isHandlerEnabled(response.config)) {
    // Handle responses
  }
  return response
}

// Enable interceptors
axiosInstance.interceptors.response.use(
  response => successHandler(response),
  error => errorHandler(error)
)
```

Q135. How to use `useSpring()` for animation?

React Spring is a spring-physics based animation library that powers most UI related animation in React. It is a bridge on the two existing React animation libraries; `React Motion` and `Animated`. It inherits animated powerful interpolations and performance while maintaining react-motion's ease of use.

There are 5 hooks in react-spring currently:

- `useSpring` a single spring, moves data from a -> b
- `useSprings` multiple springs, for lists, where each spring moves data from a -> b

- `useTrail` multiple springs with a single dataset, one spring follows or trails behind the other
- `useTransition` for mount/unmount transitions (lists where items are added/removed/updated)
- `useChain` to queue or chain multiple animations together

1. `useSpring()`

It turns defined values into animated values. It does this in two ways, either by overwriting the existing props with a different set of props on component re-render or by passing an updater function that returns a different set of props that is then used to update the props using `set`.

Example:

```
import {useSpring, animated} from 'react-spring'

function App() {
  const props = useSpring({opacity: 1, from: {opacity: 0}})
  return <animated.div style={props}>I will fade in</animated.div>
}
```

2. `useSpring()`

It works kind of like a mix between `useSpring` and `useTransition` in that it takes an array, maps over it, and uses the `from` and `to` properties to assign the animation. For our styles we can just pass in the values from each item in our array.

Example:

```
import React, { useState } from 'react'
import { animated, useSprings } from 'react-spring'
```

```
const App = () => {
  const [on, toggle] = useState(false)

  const items = [
    { color: 'red', opacity: .5 },
    { color: 'blue', opacity: 1 },
    { color: 'green', opacity: .2 },
    { color: 'orange', opacity: .8 },
  ]

  const springs = useSprings(items.length, items.map(item => ({
    from: { color: '#fff', opacity: 0 },
    to: {
      color: on ? item.color : '#fff',
      opacity: on ? item.opacity : 0
    }
  })))

  return (
    <div>
      {springs.map(animation => (
        <animated.div style={animation}>Hello World</animated.div>
      )))
    <button onClick={() => toggle(!on)}>Change</button>
  </div>
)
```

```
}
```

3. useTrail()

useTrail allows to create an effect similar to both useSpring and useSprings, it will allow us to attach an animation to multiple items but instead of being executed at the same time, they will be executed one after the other. It just takes a number for how many we want and the style object.

Example:

```
import { animated, useTrail, config } from 'react-spring'

const App = () => {
  const [on, toggle] = useState(false)

  const springs = useTrail(5, {
    to: { opacity: on ? 1 : 0 },
    config: { tension: 250 }
  })

  return (
    <div>
      {springs.map((animation, index) => (
        <animated.div style={animation} key={index}>Hello
        World</animated.div>
      ))}

      <button onClick={() => toggle(!on)}>Change</button>
    </div>
  )
}
```

4. useTransition()

useTransition allows to create an animated transition group. It takes in the elements of the list, their keys, and lifecycles. The animation is triggered on appearance and disappearance of the elements.

Example:

```
import React, { useState } from 'react'
import { animated, useTransition } from 'react-spring'

const [on, toggle] = useState(false)

const transition = useTransition(on, null, {
  from: { opacity: 0 },
  enter: { opacity: 1 },
  leave: { opacity: 0 }
})

return (
<div>
  {transition.map(({ item, key, props }) => (
    item && <animated.div style={props}>Hello world</animated.div>
  ))}

  <button onClick={() => toggle(!on)}>Change</button>
</div>
)
```

5. useChain()

useChain allows to set the execution sequence of previously defined animation hooks. To do this, you need to use `refs`, which will subsequently prevent the independent execution of the animation.

Example:

```
import React, { useState, useRef } from 'react'
import { animated, useSpring, useTrail, useChain } from 'react-spring'

const App = () => {
  const [on, toggle] = useState(false)

  const springRef = useRef()
  const spring = useSpring({
    ref: springRef,
    from: { opacity: .5 },
    to: { opacity: on ? 1 : .5 },
    config: { tension: 250 }
  })

  const trailRef = useRef()
  const trail = useTrail(5, {
    ref: trailRef,
    from: { fontSize: '10px' },
    to: { fontSize: on ? '45px' : '10px' }
  })

  useChain(on ? [springRef, trailRef] : [trailRef, springRef])

  return (
    <div>
      {trail.map((animation, index) => (
        <animated.h1 style={{ ...animation, ...spring }} key={index}>Hello
        World</animated.h1>
      ))}
      <button onClick={() => toggle(!on)}>Change</button>
    </div>
  )
}
```

Q136. How to do caching in React?

In React, caching data can be achieved in multiple ways

- Local Storage
- Redux Store
- Keep data between mounting and unmounting

1. Memoizing Fetched Data

Memoization is a technique we would use to make sure that we don't hit the API if we have made some kind of request to fetch it at some initial phase. Storing the result of expensive fetch calls will save the users some load time, therefore, increasing overall performance.

Example:

```

const cache = {}

const useFetch = (url) => {
  const [status, setStatus] = useState('idle')
  const [data, setData] = useState([])

  useEffect(() => {
    if (!url) return

    const fetchData = async () => {
      setStatus('fetching')

      if (cache[url]) {
        const data = cache[url]
        setData(data)
        setStatus('fetched')
      } else {
        const response = await fetch(url)
        const data = await response.json()
        cache[url] = data // set response in cache
        setData(data)
        setStatus('fetched')
      }
    }
  })

  fetchData()
}, [url])

return { status, data }
}

```

Here, we're mapping URLs to their data. So, if we make a request to fetch some existing data, we set the data from our local cache, else, we go ahead to make the request and set the result in the cache. This ensures we do not make an API call when we have the data available to us locally.

2. Memoizing Data With `useRef()`

With `useRef()`, we can set and retrieve mutable values at ease and its value persists throughout the component's lifecycle.

```

const useFetch = (url) => {
  const cache = useRef({})
  const [status, setStatus] = useState('idle')
  const [data, setData] = useState([])

  useEffect(() => {
    if (!url) return

    const fetchData = async () => {
      setStatus('fetching')

      if (cache.current[url]) {
        const data = cache.current[url]
        setData(data)
        setStatus('fetched')
      } else {
        const response = await fetch(url)
        const data = await response.json()

```

```

        cache.current[url] = data // set response in cache
        setData(data)
        setStatus('fetched')
    }
}

fetchData()
, [url])

return { status, data }
}

```

3. Using localStorage()

```

const InitialState = {
    someState: 'a'
}
class App extends Component {

constructor(props) {
    super(props)

    // Retrieve the last state
    this.state = localStorage.getItem("appState") ?
    JSON.parse(localStorage.getItem("appState")) : InitialState
}

componentWillUnmount() {
    // Remember state for the next mount
    localStorage.setItem('appState', JSON.stringify(this.state))
}

render() {
    ...
}
}

```

export default App

4. Keep data between mounting and unmounting

```

import React, { Component } from 'react'

// Set initial state
let state = { counter: 5 }

class Counter extends Component {

constructor(props) {
    super(props)

    // Retrieve the last state
    this.state = state

    this.onClick = this.onClick.bind(this)
}

```

```

componentWillUnmount() {
  // Remember state for the next mount
  state = this.state
}

onClick(e) {
  e.preventDefault()
  this.setState(prev => ({ counter: prev.counter + 1 }))
}

render() {
  return (
    <div>
      <span>{ this.state.counter }</span>
      <button onClick={this.onClick}>Increase</button>
    </div>
  )
}
}

export default Counter

```

Q137. What is `useRef()` in React?

Q138. What is `useHooks()` in React?

React Unit Testing

Q1. Explain react unit testing using Jest and Enzyme?

Jest

Jest is a JavaScript unit testing framework, used by Facebook to test services and React applications. Jest acts as a **test runner**, **assertion library**, and **mocking library**. Jest also provides Snapshot testing, the ability to create a rendered *snapshot* of a component and compare it to a previously saved *snapshot*. The test will fail if the two do not match.

Enzyme

Enzyme is a JavaScript Testing utility for React that makes it easier to assert, manipulate, and traverse your React Components output. Enzyme, created by Airbnb, adds some great additional utility methods for rendering a component (or multiple components), finding elements, and interacting with elements.

Setup with Create React App

```

# for rendering snapshots
npm install react-test-renderer --save-dev

# for dom testing
npm install enzyme --save-dev
{
  "react": "^16.13.1",
  "@testing-library/jest-dom": "^4.2.4",
  "@testing-library/react": "^9.5.0",
  "@testing-library/user-event": "^7.2.1",
  "enzyme": "3.9",
  "jest": "24.5.0",

```

```
"jest-cli": "24.5.0",
"babel-jest": "24.5.0"
}

Set up a React application
npx create-react-app counter-app
// src/App.js

import React, { Component } from 'react'

class App extends Component {
  constructor() {
    super()
    this.state = {
      count: 0,
    }
  }
  makeIncrementer = amount => () =>
    this.setState(prevState => ({
      count: prevState.count + amount,
    }))
  increment = this.makeIncrementer(1)
  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button className="increment" onClick={this.increment}>Increment
        count</button>
      </div>
    )
  }
}
export default App
```

Using Enzyme

```
// src/App.test.js

import React from 'react'
import { shallow } from 'enzyme'
import App from './App'

describe('App component', () => {
  it('starts with a count of 0', () => {
    const wrapper = shallow(<App />)
    const text = wrapper.find('p').text()
    expect(text).toEqual('Count: 0')
  })
})
```

Testing User Interaction

```
// src/App.test.js

describe('App component', () => {

  it('increments count by 1 when the increment button is clicked', () => {
    const wrapper = shallow(<App />)
    const incrementBtn = wrapper.find('button.increment')
```

```

incrementBtn.simulate('click')
const text = wrapper.find('p').text()
expect(text).toEqual('Count: 1')
})
})

```

- [Jest Framework](#)
- [Enzyme Framework](#)

Q2. Explain about shallow() method in enzyme?

The `shallow()` method is used to render the single component that we are testing. It does not render child components. Simple shallow calls the `constructor()`, `render()`, `componentDidMount()` methods.

Example:

```

import React from "react"
import { shallow } from "enzyme"
import Enzyme from "enzyme"
import Adapter from "enzyme-adapter-react-16"

Enzyme.configure({ adapter: new Adapter() })

function Name(props) {
  return <span>Welcome {props.name}</span>
}

function Welcome(props) {
  return (
    <h1>
      <Name name={props.name} />
    </h1>
  )
}

const wrapper = shallow(<Welcome name="Alex" />)
console.log(wrapper.debug())

```

When not to use Shallow Rendering

- Shallow rendering is not useful for testing the end-user experience.
- Shallow rendering is not useful for testing DOM rendering or interactions.
- Shallow rendering is not useful for integration testing.
- Shallow rendering is not useful for browser testing.
- Shallow rendering is not useful for end to end testing.

Q3. What is mount() method in Enzyme?

Full DOM rendering generates a virtual DOM of the component with the help of a library called `jsdom`. It is useful when we want to test the behavior of a component with its children. This is more suitable when there are components which directly interfere with DOM API or lifecycle methods of React. Simple mount calls the `constructor()`, `render()`, `componentDidMount()` methods.

Example:

```

...
import ListItem from './ListItem'
...

return (
  <ul className="list-items">
    {items.map(item => <ListItem key={item} item={item} />) }
  </ul>
)
import React from 'react'
import { mount } from '../enzyme'
import List from './List'

describe('List tests', () => {

  it('renders list-items', () => {
    const items = ['one', 'two', 'three']

    const wrapper = mount(<List items={items} />)

    // Let's check what wrong in our instance
    console.log(wrapper.debug())

    // Expect the wrapper object to be defined
    expect(wrapper.find('.list-items')).toBeDefined()
    expect(wrapper.find('.item')).toHaveLength(items.length)
  })

  ...
})

```

Q4. What is `render()` method in Enzyme?

Static rendering is used to render react components to static HTML. It's implemented using a library called **Cheerio**. It renders the children. But this does not have access to React lifecycle methods.

For static rendering, we can not access to Enzyme API methods such as `contains()` and `debug()`. However we can access to the full arsenal of Cheerios manipulation and traversal methods such as `addClass()` and `find()` respectively.

Example:

```

import React from 'react'
import { render } from '../enzyme'

import List from './List'
import { wrap } from 'module'

describe('List tests', () => {

  it('renders list-items', () => {
    const items = ['one', 'two', 'three']
    const wrapper = render(<List items={items} />)

    wrapper.addClass('foo')
  })
})

```

```

    // Expect the wrapper object to be defined
    expect(wrapper.find('.list-items')).toBeDefined()
    expect(wrapper.find('.item')).toHaveLength(items.length)
  })

  ...
})

```

Q5. What is the purpose of the ReactTestUtils package?

ReactTestUtils is used to test React-based components. It can simulate all the JavaScript-based events, which ReactJS supports. Some of its frequently methods are

- `act()`
- `mockComponent()`
- `isElement()`
- `isElementOfType()`
- `isDOMComponent()`
- `renderIntoDocument()`
- `Simulate()`

act()

To prepare a component for assertions, wrap the code rendering it and performing updates inside an `act()` call. This makes your test run closer to how React works in the browser.

```

class Counter extends React.Component {
  constructor(props) {
    super(props)
    this.state = {count: 0}
    this.handleClick = this.handleClick.bind(this)
  }
  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`
  }
  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`
  }
  handleClick() {
    this.setState(state => ({
      count: state.count + 1,
    }))
  }
  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={this.handleClick}>
          Click me
        </button>
      </div>
    )
  }
}

import React from 'react'
import ReactDOM from 'react-dom'
import { act } from 'react-dom/test-utils'

```

```

import Counter from './Counter'

let container

beforeEach(() => {
  container = document.createElement('div')
  document.body.appendChild(container)
})

afterEach(() => {
  document.body.removeChild(container)
  container = null
})

it('can render and update a counter', () => {
  // Test first render and componentDidMount
  act(() => {
    ReactDOM.render(<Counter />, container)
  })
  const button = container.querySelector('button')
  const label = container.querySelector('p')
  expect(label.textContent).toBe('You clicked 0 times')
  expect(document.title).toBe('You clicked 0 times')

  // Test second render and componentDidUpdate
  act(() => {
    button.dispatchEvent(new MouseEvent('click', {bubbles: true}))
  })
  expect(label.textContent).toBe('You clicked 1 times')
  expect(document.title).toBe('You clicked 1 times')
})
}

```

Q6. What is react-test-renderer package in React?

This package provides a React renderer that can be used to render React components to pure JavaScript objects, without depending on the DOM or a native mobile environment.

Essentially, this package makes it easy to grab a snapshot of the platform view hierarchy (similar to a DOM tree) rendered by a React DOM or React Native component without using a browser or jsdom.

Example:

```

import React from 'react'
import renderer from 'react-test-renderer'
import App from './app.js' // The component being tested

/**
 * Snapshot tests are useful when UI does not change frequently.
 *
 * A typical snapshot test case for a mobile app renders a UI component,
 * takes a snapshot,
 * then compares it to a reference snapshot file stored alongside the test.
 */
describe('APP Component', () => {

  test('Matches the snapshot', () => {

```

```

        const tree = renderer.create(<App />).toJSON()
        expect(tree).toMatchSnapshot()
    })
}

```

Q7. Why should we use Test-Driven Development (TDD) for ReactJS?

Test-driven development is an approach when developers create a product backwards. TDD requires developers to write tests first and only then start to write the code. TDD is a development method that utilizes repetition of a short development cycle called Red-Green-Refactor.

Process:

1. Add a test
2. Run all tests and see if the new test fails (red)
3. Write the code to pass the test (green)
4. Run all tests
5. Refactor
6. Repeat

Pros:

1. Design before implementation
2. Helps prevent future regressions and bugs
3. Increases confidence that the code works as expected

Cons:

1. Takes longer to develop (but it can save time in the long run)
2. Testing edge cases is hard
3. Mocking, faking, and stubbing are all even harder

Q8. What are the benefits of using data-test selector over className or Id selector in Jest?

HTML structure and css classes tend to change due to design changes. Which will cause to rewrite tests quite often. Also, if we are using css-modules we can not rely on class names.

Because of that, React provides `data-test` attribute for selecting elements in jsx.

```

// APP Component
import React from 'react'
import './App.scss'

function App() {
  return (
    <div data-test='app-header'>
      Hello React
    </div>
  )
}
export default App
import React from 'react'
import { shallow } from 'enzyme'

```

```

import App from './App'

describe('APP Component', () => {

  test('title', () => {
    let wrapper = shallow(<APP />)
    let title = wrapper.find(`[data-test='app-header']`).text()

    expect(title).toMatch('Hello React')
  })
})

```

Redux Questions

1Q. *What is Redux?*

Redux is a predictable state container for JavaScript applications. It helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test.

Simply put, Redux is a state management tool. While it is mostly used with React, it can be used with any other JavaScript framework or library. With Redux, the state of your application is kept in a store, and each component can access any state that it needs from this store.

How Redux works

There is a central store that holds the entire state of the application. Each component can access the stored state without having to send down props from one component to another. There are three building parts: actions, store, and reducers.

Benefits and limitations of Redux

1. State transfer

State is stored together in a single place called the ‘store.’ While you do not need to store all the state variables in the ‘store,’ it is especially important to when state is being shared by multiple components or in a more complex architecture. It also allows you to call state data from any component easily.

2. Predictability

Redux is “a predictable state container for Javascript apps.” Because reducers are pure functions, the same result will always be produced when a state and action are passed in.

3. Maintainability

Redux provides a strict structure for how the code and state should be managed, which makes the architecture easy to replicate and scale for somebody who has previous experience with Redux.

4. Ease of testing and debugging

Redux makes it easy to test and debug your code since it offers powerful tools such as Redux DevTools in which you can time travel to debug, track your changes, and much more to streamline your development process.

2Q. *Explain pros and cons of Redux?*

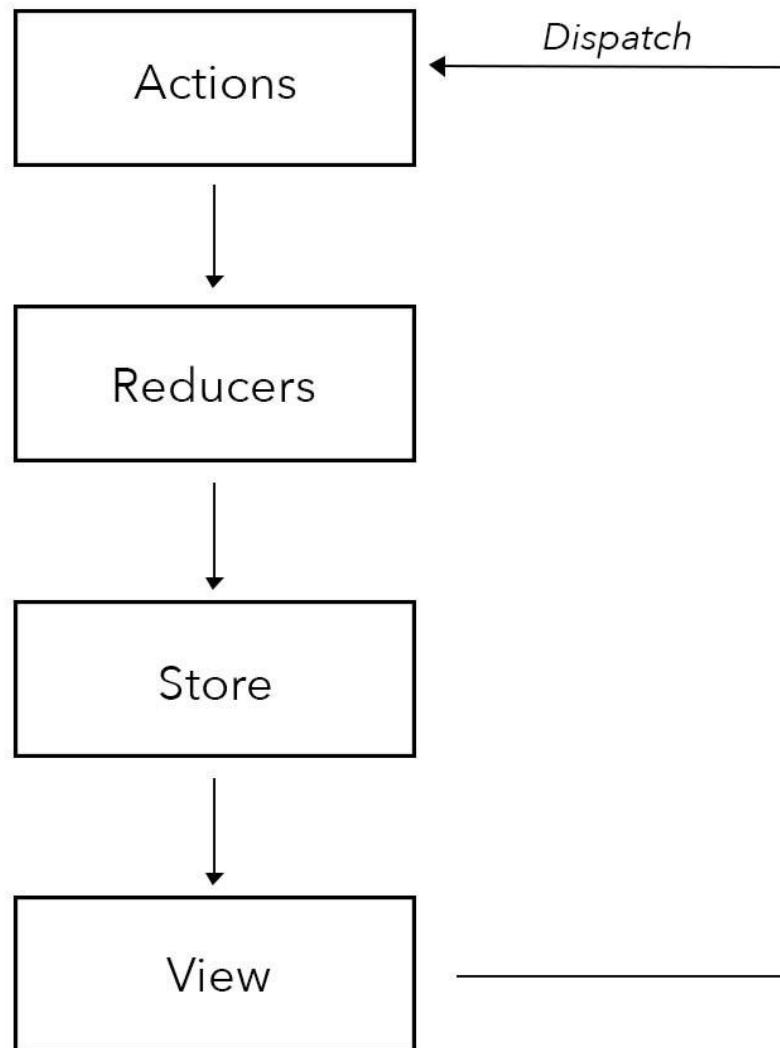
Pros using redux

- Central store, any component can access any state from the store, there is no need of passing props back and forth.
- Another way to look at centralised store, it persists the state of a component even after the component has unmounted.
- Prevents unnecessary re-renders, as when the state changes it returns new state which uses shallow copy.
- Testing will be easy as UI and data management are separated.
- History of state is maintained which helps in implementing features like undo very easily.

Cons using redux

- No encapsulation. Any component can access the data which can cause security issues.
- Boilerplate code. Restricted design.
- As state is immutable in redux, the reducer updates the state by returning a new state every time which can cause excessive use of memory.

3Q. What are redux core concepts?



1. Actions in Redux

Action is static information about the event that initiates a state change. When you update your state with Redux, you always start with an action. Actions are in the form of Javascript objects,

containing a `type` and an optional `payload`. Actions are sent using the `store.dispatch()` method. Actions are created via an action creator.

Action creators are simple functions that help to create actions. They are functions that return action objects, and then, the returned object is sent to various reducers in the application.

Example:

```
const setLoginStatus = (name, password) => {
  return {
    type: "LOGIN",
    payload: {
      username: "foo",
      password: "bar"
    }
  }
}
```

2. Reducers in Redux

Reducers are pure functions that take the current state of an application, perform an action, and return a new state. These states are stored as objects, and they specify how the state of an application changes in response to an action sent to the store.

It is based on the `reduce` function in JavaScript, where a single value is calculated from multiple values after a callback function has been carried out.

```
const LoginComponent = (state = initialState, action) => {
  switch (action.type) {

    // This reducer handles any action with type "LOGIN"
    case "LOGIN":
      return state.map(user => {
        if (user.username !== action.username) {
          return user
        }

        if (user.password == action.password) {
          return {
            ...user,
            login_status: "LOGGED IN"
          }
        }
      });
    default:
      return state;
  }
}
```

combine multiple reducers: The `combineReducers()` helper function turns an object whose values are different reducing functions into a single reducing function you can pass to `createStore`.

Syntax

```
const rootReducer = combineReducers(reducer1, reducer2)
```

3. Store in Redux

A Store is an object that holds the whole state tree of your application. The Redux store is the application state stored as objects. Whenever the store is updated, it will update the React components subscribed to it. The store has the responsibility of storing, reading, and updating state.

Example:

```
import React from 'react'
import { render } from 'react-dom'
import { Provider } from 'react-redux'
import { createStore } from 'redux'
import rootReducer from './reducers'
import App from './components/App'

const store = createStore(rootReducer)
render (
  <provider store="{store}">
    <app>
  </app></provider>,
  document.getElementById('root')
)
```

When using Redux with React, states will no longer need to be lifted up; thus, it makes it easier to trace which action causes any change.

4. Dispatch

Dispatch is a method that triggers an action with type and payload to Reducer.

```
store.dispatch()
```

5. Subscribe

Subscribe is a method that is used to subscribe data/state from the Store.

```
store.subscribe()
```

6. Provider

The Provider is a component that has a reference to the Store and provides the data from the Store to the component it wraps.

7. Connect

Connect is a function that communicates with the Provider.

8. Middleware

Middleware is the suggested way to extend Redux with custom functionality. Middlewares are used to dispatch async functions. We configure Middleware's while creating a store.

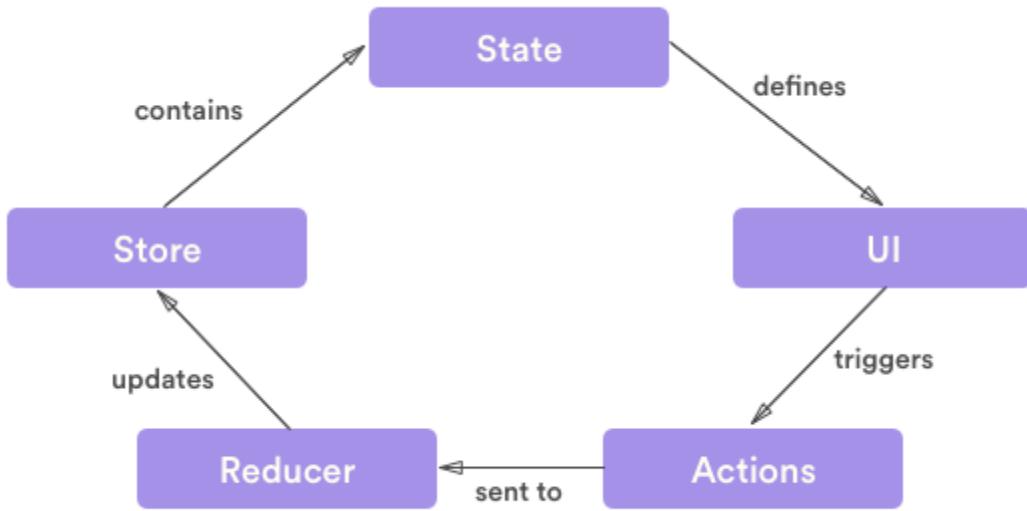
Syntax

```
const store = createStore(reducers, initialState, middleware);
```

4Q. What do you understand by "Single source of truth" in Redux?

The single source of truth is our state tree, that is not rewritten or reshaped. It gives us the availability to easily retrieve information in constant time and maintain a clean structure for the state of our application.

In React-Redux applications, when your Redux is a single source of truth, it means that the only way to change your data in UI is to dispatch redux action which will change state within redux reducer. And your React components will watch this reducer and if that reducer changes, then UI will change itself too. But never other way around, because Redux state is single source of truth.



A practical example would be that you have Redux store which contains items you want to display. In order to change list of items to be displayed, you don't change this data anywhere else other than store. And if that is changed, everything else related to it, should change as well.

5Q. What are the features of Workflow in Redux?

When using Redux with React, states will no longer need to be lifted up. Everything is handled by Redux. Redux simplifies the app and makes it easier to maintain.

- Redux offers a solution for storing all your application state in one place, called a **store**.
- Components then **dispatch** state changes to the store, not directly to other components.
- The components that need to be aware of state changes can subscribe to the store.
- The **store** can be thought of as a "middleman" for all state changes in the application.
- With Redux involved, components don't communicate directly with each other. Rather, all state changes must go through the single source of truth, the **store**.

Core Principal

Redux has three core principals:

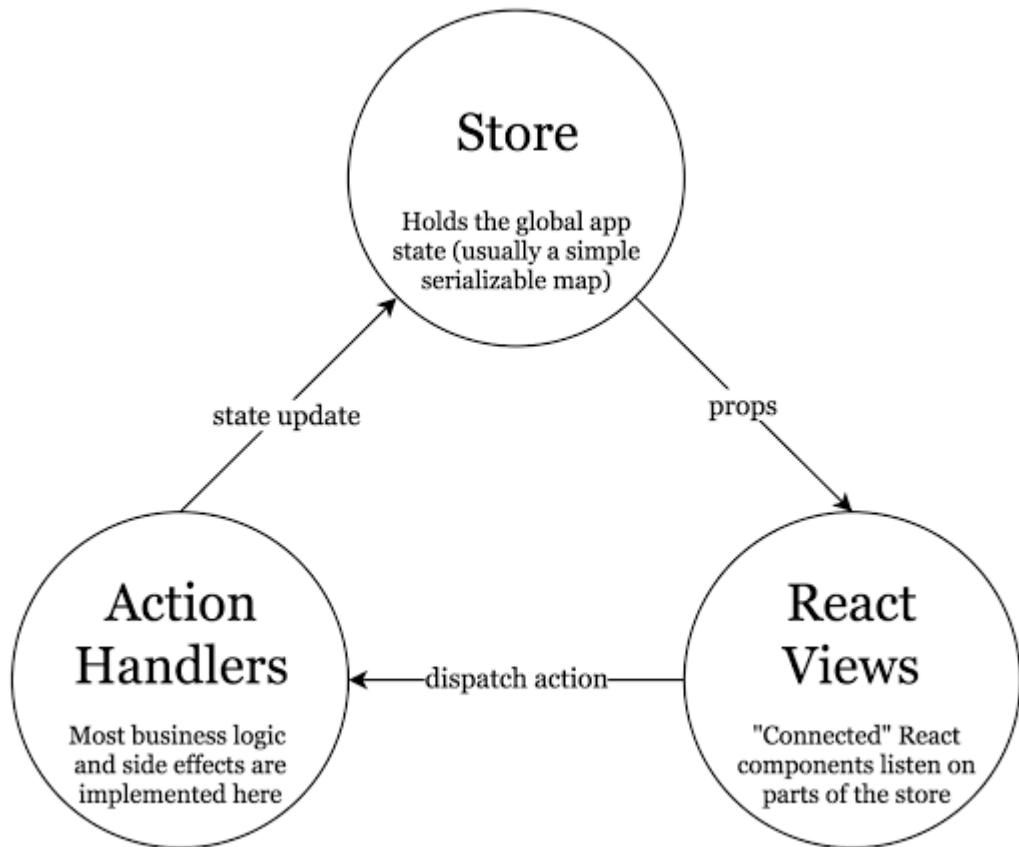
- 1. Single Source of Truth:** The state of your whole application is stored in an object tree within a single **store**.
- 2. State Is Read-Only:** The only way to change the state is to dispatch an **action**, an object describing what happened.
- 3. Changes Are Made With Pure Functions:** To specify how the state tree is transformed by actions, you write pure **reducers**.

Redux Workflow

Redux allows you to manage the state of the application using Store. A child component can directly access the state from the Store.

The following are details of how Redux works:

- When UI Event triggers (OnClick, OnChange, etc) it can dispatch Actions based on the event.
- Reducers process Actions and return a new state as an Object.
- The new state of the whole application goes into a single Store.
- Components can easily subscribe to the Store.



6Q. What is Redux Thunk used for?

Redux Thunk is a **middleware** that lets you call action creators that return a function instead of an action object. That function receives the store's `dispatch` method, which is then used to dispatch regular synchronous actions inside the body of the function once the asynchronous operations have completed. The inner function receives the store methods `dispatch()` and `getState()` as parameters.

Setup

```

# install create react app
npm install -g create-react-app

# Create a React App
create-react-app my-simple-async-app

# Switch directory
cd my-simple-app

# install Redux-Thunk
npm install --save redux react-redux redux-thunk

```

Example:

We are going to use Redux Thunk to asynchronously fetch the most recently updated repos by username from Github using this REST URL:

<https://api.github.com/users/learning-zone/repos?sort=updated>

```
import { applyMiddleware, combineReducers, createStore } from 'redux'
```

```

import thunk from 'redux-thunk'

// actions.js
export const addRepos = repos => ({
  type: 'ADD_REPOS',
  repos,
})

export const clearRepos = () => ({ type: 'CLEAR_REPOS' })

export const getRepos = username => async dispatch => {
  try {
    const url = `https://api.github.com/users/${username}/repos?sort=updated`
    const response = await fetch(url)
    const responseBody = await response.json()
    dispatch(addRepos(responseBody))
  } catch (error) {
    console.error(error)
    dispatch(clearRepos())
  }
}

// reducers.js
export const repos = (state = [], action) => {
  switch (action.type) {
    case 'ADD_REPOS':
      return action.repos
    case 'CLEAR_REPOS':
      return []
    default:
      return state
  }
}

export const reducers = combineReducers({ repos })

// store.js
export function configureStore(initialState = {}) {
  const store = createStore(reducers, initialState, applyMiddleware(thunk))
  return store
}

export const store = configureStore()

applyMiddleware(thunk): This tells redux to accept and execute functions as return values.
Redux usually only accepts objects like { type: 'ADD_THINGS', things: ['list', 'of', 'things'] }.
The middleware checks if the action's return value is a function and if it is it will execute the
function and inject a callback function named dispatch. This way you can start an asynchronous
task and then use the dispatch callback to return a regular redux object action some time in the
future.

// This is your typical redux sync action
function syncAction(listOfThings) {
  return { type: 'ADD_THINGS', things: listOfThings }
}

```

```

// This would be the async version
// where we may need to go fetch the
// list of things from a server before
// adding them via the sync action
function asyncAction() {
  return function(dispatch) {
    setTimeout(function() {
      dispatch(syncAction(['list', 'of', 'things']))
    }, 1000)
  }
}

```

App.js

```

import React, { Component } from 'react'

import { connect } from 'react-redux'

import { getRepos } from './redux'

// App.js
export class App extends Component {
  state = { username: 'learning-zone' }

  componentDidMount() {
    this.updateRepoList(this.state.username)
  }

  updateRepoList = username => this.props.getRepos(username)

  render() {
    return (
      <div>
        <h1>I AM AN ASYNC APP!!!</h1>

        <strong>Github username: </strong>
        <input
          type="text"
          value={this.state.username}
          onChange={ev => this.setState({ username: ev.target.value })}
          placeholder="Github username..." />
        <button onClick={() => this.updateRepoList(this.state.username)}>
          Get Lastest Repos
        </button>

        <ul>
          {this.props.repos.map((repo, index) => (
            <li key={index}>
              <a href={repo.html_url} target="_blank">
                {repo.name}
              </a>
            </li>
          )))
        </ul>
      </div>
    )
  }
}

```

```

        )
    }
}

// AppContainer.js
const mapStateToProps = (state, ownProps) => ({ repos: state.repos })
const mapDispatchToProps = { getRepos }
const AppContainer = connect(mapStateToProps, mapDispatchToProps)(App)

export default AppContainer
index.js
import React from 'react'
import ReactDOM from 'react-dom'
import AppContainer from './App'
import './index.css'

// Add these imports - Step 1
import { Provider } from 'react-redux'
import { store } from './redux'

// Wrap existing app in Provider - Step 2
ReactDOM.render(
  <Provider store={store}>
    <AppContainer />
  </Provider>,
  document.getElementById('root')
)

```

7Q. What is difference between presentational component and container component in react redux?

1. Container Components

- Container components are primarily concerned with how things work
- They rarely have any HTML tags of their own, aside from a wrapping `<div>`
- They are often stateful
- They are responsible for providing data and behavior to their children (usually presentational components)

Container is an informal term for a React component that is `connect`-ed to a redux store.

Containers receive Redux state updates and `dispatch` actions, and they usually don't render DOM elements; they delegate rendering to **presentational** child components.

Example:

```

class Collage extends Component {
  constructor(props) {
    super(props);

    this.state = {
      images: []
    };
  }
  componentDidMount() {
    fetch('/api/current_user/image_list')
      .then(response => response.json())

```

```

        .then(images => this.setState({images}));
    }
    render() {
      return (
        <div className="image-list">
          {this.state.images.map(image => {
            <div className="image">
              <img src={book.image_url} />
            </div>
          })}
        </div>
      )
    }
  }
}

```

2. Presentational Components

- Presentational Components are primarily concerned with how things look
- Probably only contain a render method and little else logic
- They do not know how to load or alter the data that they render
- They are best written as stateless functional components

Example:

```

//defining the component as a React Component
class Image extends Component {
  render() {
    return <img src={this.props.image} />;
  }
}
export default Image
//defining the component as a constant
const Image = props => (
  <img src={props.image} />
)
export default Image

```

8Q. Explain the role of Reducer?

A reducer is a function that determines changes to an application's state. It uses the action it receives to determine this change. Redux manage an application's state changes in a single store so that they behave consistently. Redux relies heavily on reducer functions that take the previous state and an action in order to execute the next state.

1. State

State changes are based on a user's interaction, or even something like a network request. If the application's state is managed by Redux, the changes happen inside a reducer function — this is the only place where state changes happen. The reducer function makes use of the initial state of the application and something called action, to determine what the new state will look like.

Syntax

```

const contactReducer = (state = initialState, action) => {
  // Do something
}

```

2. State Parameter

The state parameter that gets passed to the reducer function has to be the current state of the application. In this case, we're calling that our initialState because it will be the first (and current) state and nothing will precede it.

```
contactReducer(initialState, action)
```

Example:

Let's say the initial state of our app is an empty list of contacts and our action is adding a new contact to the list.

```
const initialState = {  
  contacts: []  
}
```

3. Action Parameter

An action is an object that contains two keys and their values. The state update that happens in the reducer is always dependent on the value of action.type.

```
const action = {  
  type: 'NEW_CONTACT',  
  name: 'Alex K',  
  location: 'Lagos Nigeria',  
  email: 'alex@example.com'  
}
```

There is typically a payload value that contains what the user is sending and would be used to update the state of the application. It is important to note that `action.type` is required, but `action.payload` is optional. Making use of `payload` brings a level of structure to how the action object looks like.

4. Updating State

The state is meant to be immutable, meaning it shouldn't be changed directly. To create an updated state, we can make use of `Object.assign()` or opt for the **spread operator**.

Example:

```
const contactReducer = (state, action) => {  
  switch (action.type) {  
    case 'NEW_CONTACT':  
      return {  
        ...state, contacts:  
        [...state.contacts, action.payload]  
      }  
    default:  
      return state  
  }  
}
```

This ensures that the incoming state stays intact as we append the new item to the bottom.

```
const initialState = {  
  contacts: [{  
    name: 'Alex K',  
    age: 26  
  }]  
}  
  
const contactReducer = (state = initialState, action) => {  
  switch (action.type) {  
    case "NEW_CONTACT":  
      return Object.assign({}, state, {  
        contacts: [...state.contacts, action.payload]  
      })  
    default:  
      return state  
  }  
}
```

```
        });
    default:
        return state
    }
}

class App extends React.Component {
constructor(props) {
    super(props)
    this.name = React.createRef()
    this.age = React.createRef()
    this.state = initialState
}

handleSubmit = e => {
    e.preventDefault()
    const action = {
        type: "NEW_CONTACT",
        payload: {
            name: this.name.current.value,
            age: this.age.current.value
        }
    }
    const newState = contactReducer(this.state, action)
    this.setState(newState)
}

render() {
    const { contacts } = this.state
    return (
        <div className="box">
            <div className="content">
                <pre>{JSON.stringify(this.state, null, 2)}</pre>
            </div>

            <div className="field">
                <form onSubmit={this.handleSubmit}>
                    <div className="control">
                        <input className="input" placeholder="Full Name" type="text"
ref={this.name} />
                    </div>
                    <div className="control">
                        <input className="input" placeholder="Age" type="number"
ref={this.age} />
                    </div>
                    <div>
                        <button type="submit" className="button">Submit</button>
                    </div>
                </form>
            </div>
        </div>
    )
}
}
```

```
ReactDOM.render(  
  <App />,  
  document.getElementById('root')  
)
```

9Q. What are Pure Functions and why should the reducer be a "pure" function?

Pure Functions

Any function that doesn't alter input data and that doesn't depend on external state (like a database, DOM, or global variable) and consistently provides the same output for the same input is a "pure" function.

A pure function adheres to the following rules:

- A function returns the same result for same arguments.
- Its evaluation has no side effects, i.e., it does not alter input data.
- No mutation of local & global variables.
- It does not depend on the external state like a global variable.

Example:

The below `add()` function doesn't alter "a" or "b", doesn't depending on external state, and always returns the same output for the same input.

```
const add = (a, b) => a + b //pure function
```

Why Reducer must be pure function

Redux takes a given state (object) and passes it to each reducer in a loop. And it expects a brand new object from the reducer if there are any changes. And it also expects to get the old object back if there are no changes.

Redux simply checks whether the old object is the same as the new object by comparing the memory locations of the two objects. So if you mutate the old object's property inside a reducer, the "new state" and the "old state" will both point to the same object. Hence Redux thinks nothing has changed! So this won't work.

10Q. How to split the reducers?

Putting all your update logic into a single reducer function is quickly going to become unmaintainable. While there's no single rule for how long a function should be, it's generally agreed that functions should be relatively short and ideally only do one specific thing. It's good programming practice to take pieces of code that are very long or do many different things, and break them into smaller pieces that are easier to understand.

In Redux reducer, we can split some of our reducer logic out into another function, and call that new function from the parent function. These new functions would typically fall into one of three categories:

1. Small utility functions containing some reusable chunk of logic that is needed in multiple places (which may or may not be actually related to the specific business logic)
2. Functions for handling a specific update case, which often need parameters other than the typical (state, action) pair
3. Functions which handle all updates for a given slice of state. These functions do generally have the typical (state, action) parameter signature

These terms will be used to distinguish between different types of functions and different use cases:

- **reducer**: any function with the signature `(state, action) -> newState` (ie, any function that could be used as an argument to `Array.prototype.reduce`)
- **root reducer**: the reducer function that is actually passed as the first argument to `createStore`. This is the only part of the reducer logic that must have the `(state, action) -> newState` signature.
- **slice reducer**: a reducer that is being used to handle updates to one specific slice of the state tree, usually done by passing it to `combineReducers`
- **case function**: a function that is being used to handle the update logic for a specific action. This may actually be a reducer function, or it may require other parameters to do its work properly.
- **higher-order reducer**: a function that takes a reducer function as an argument, and/or returns a new reducer function as a result (such as `combineReducers`, or `redux-undo`).

Benefits

- **For fast page loads**

Splitting reducers will have an advantage of loading only required part of web application which in turn makes it very efficient in rendering time of main pages

- **Organization of code**

Splitting reducers on page level or component level will give a better code organization instead of just putting all reducers at one place. Since reducer is loaded only when page/component is loaded will ensure that there are standalone pages which are not dependent on other parts of the application.

- **One page/component**

One reducer design pattern. Things are better written, read and understood when they are modular. With dynamic reducers, it becomes possible to achieve it.

- **SEO**

With reducer level code-splitting, reducers can be code split on a split component level which will reduce the loading time of website thereby increasing SEO rankings.

11Q. *How to create action creators react with redux?*

Action Type

An action type is a string that simply describes the type of an action. They're commonly stored as constants or collected in enumerations to help reduce typos.

Example:

```
export const Actions = {
  GET_USER_DETAILS_REQUEST: 'GET_USER_DETAILS_REQUEST',
  GET_USER_DETAILS_SUCCESS: 'GET_USER_DETAILS_SUCCESS',
  GET_USER_DETAILS_FAILURE: 'GET_USER_DETAILS_FAILURE',
  ...
}
```

Action

An action is like a message that we send (i.e. dispatch) to our central Redux store. It can literally be anything. But ideally we want to stick to an agreed-upon pattern. And the standard pattern is as follows (this is a TypeScript type declaration):

```
type Action = {
```

```
    type: string;      // Actions MUST have a type
    payload?: any;    // Actions MAY have a payload
    meta?: any;       // Actions MAY have meta information
    error?: boolean; // Actions MAY have an error field
                      // when true, payload SHOULD contain an Error
}
```

An action to fetch the user named "Alex" might look something like this

```
{
  type: 'GET_USER_DETAILS_REQUEST',
  payload: 'Alex'
}
```

Action Creator

```
export const getUserDetailsRequest = id => ({
  type: Actions.GET_USER_DETAILS_REQUEST,
  payload: id,
})
```

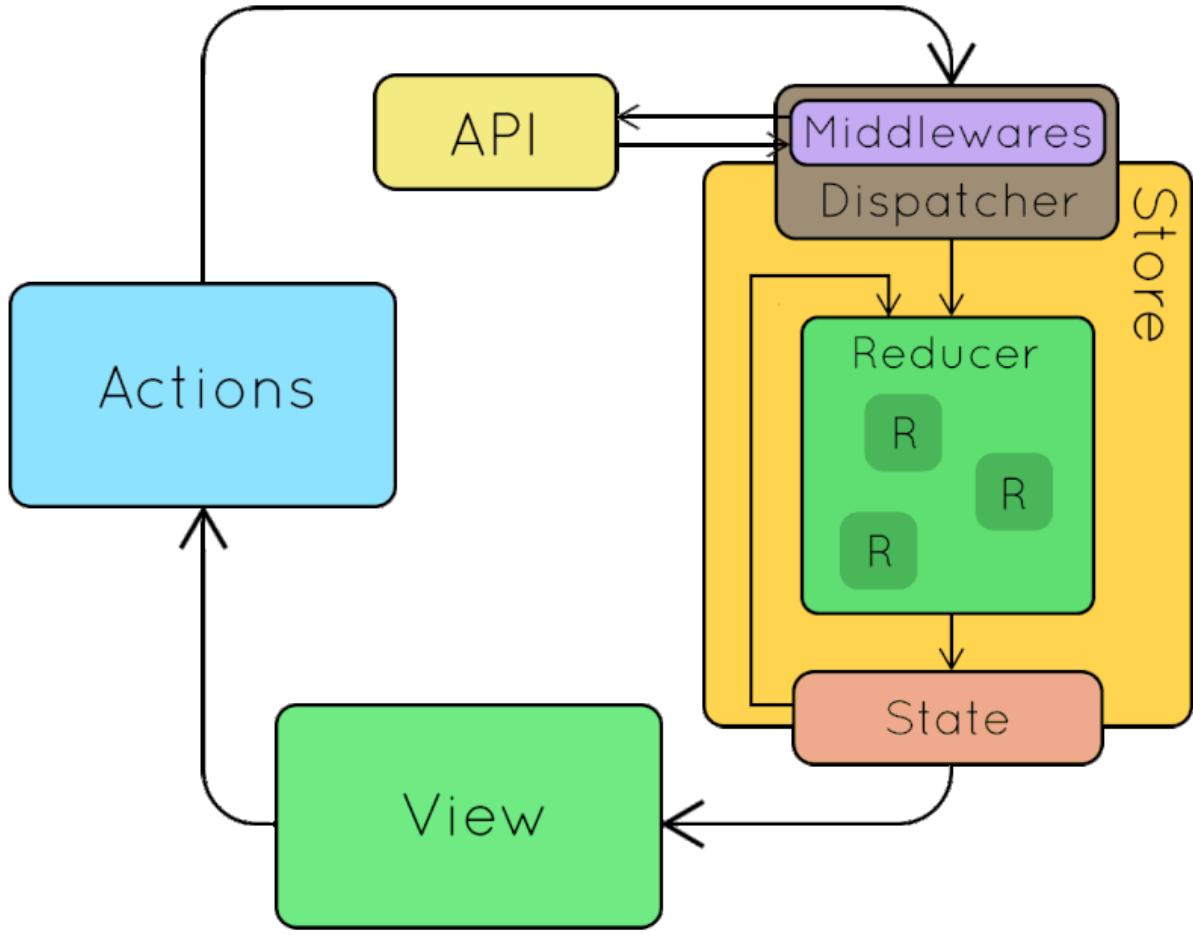
When writing basic Redux, an action creator simply returns an action. You would typically dispatch the action to your store immediately.

```
store.dispatch(getUserDetailsRequest('Alex'))
```

Although, realistically, you'll be doing this via dispatch properties that are passed into a React component like this:

```
// ES6
export const mapDispatchToProps = dispatch => ({
  onClick: () => dispatch(getUserDetailsRequest('Alex'))
})
```

12Q. *How to set the dataflow using react with redux?*



Redux offers this data sharing of components possible by maintaining one single state in the store. A single source of truth. All the components which want to get state data at some point are subscribed to the store and they will receive the state each time it gets updated.

Redux has five main entities. Action Creators, Dispatching Function, Reducers, State and Store.

- An action is dispatched when a user interacts with the application.
- The root reducer function is called with the current state and the dispatched action. The root reducer may divide the task among smaller reducer functions, which ultimately returns a new state.
- The store notifies the view by executing their callback functions.
- The view can retrieve updated state and re-render again.

13Q. What are the three principles that Redux follows?

Redux can be described in three fundamental principles:

1. Single source of truth

The state of your whole application is stored in an object tree inside a single store.

This makes it easy to create universal apps, as the state from your server can be serialized and hydrated into the client with no extra coding effort. A single state tree also makes it easier to debug or inspect an application; it also enables you to persist your app's state in development, for a faster development cycle.

Example:

```
console.log(store.getState())
```

```

/* Prints
{
  visibilityFilter: 'SHOW_ALL',
  todos: [
    {
      text: 'Consider using Redux',
      completed: true,
    },
    {
      text: 'Keep all state in a single tree',
      completed: false
    }
  ]
}
*/

```

2. State is read-only

The only way to change the state is to emit an action, an object describing what happened. This ensures that neither the views nor the network callbacks will ever write directly to the state. Instead, they express an intent to transform the state. Because all changes are centralized and happen one by one in a strict order, there are no subtle race conditions to watch out for.

Example:

```

store.dispatch({
  type: 'COMPLETE_TODO',
  index: 1
})

store.dispatch({
  type: 'SET_VISIBILITY_FILTER',
  filter: 'SHOW_COMPLETED'
})

```

3. Changes are made with pure functions

To specify how the state tree is transformed by actions, you write pure reducers. Reducers are just pure functions that take the previous state and an action, and return the next state. Remember to return new state objects, instead of mutating the previous state. You can start with a single reducer, and as your app grows, split it off into smaller reducers that manage specific parts of the state tree.

```

import { combineReducers, createStore } from 'redux'

function visibilityFilter(state = 'SHOW_ALL', action) {
  switch (action.type) {
    case 'SET_VISIBILITY_FILTER':
      return action.filter
    default:
      return state
  }
}

function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return [
        ...state,
        {
          text: action.text,
          completed: false
        }
      ]
    case 'COMPLETE_TODO':
      return state.map(todo => todo.id === action.id ? { ...todo, completed: true } : todo)
    case 'SET_VISIBILITY_FILTER':
      return state.filter(todo => action.filter === 'SHOW_COMPLETED' || !todo.completed)
  }
}

```

```

    ...state,
  [
    {
      text: action.text,
      completed: false
    }
  ]
  case 'COMPLETE_TODO':
    return state.map((todo, index) => {
      if (index === action.index) {
        return Object.assign({}, todo, {
          completed: true
        })
      }
      return todo
    })
  default:
    return state
  }
}

const reducer = combineReducers({ visibilityFilter, todos })
const store = createStore(reducer)

```

14Q. How can I represent "side effects" such as AJAX calls? Why do we need things like "action creators", "thunks", and "middleware" to do async behavior?

Any meaningful web app needs to execute complex logic, usually including asynchronous work such as making AJAX requests. That code is no longer purely a function of its inputs, and the interactions with the outside world are known as "side effects".

Redux is inspired by functional programming, and out of the box, has no place for side effects to be executed. In particular, reducer functions must always be pure functions of `(state, action) => newState`. However, Redux's middleware (eg. **Redux Thunk**, **Redux Saga**) makes it possible to intercept dispatched actions and add additional complex behavior around them, including side effects.

15Q. What is the '@' (at symbol) in the Redux @connect decorator?

Decorators make it possible to annotate and modify classes and properties at design time.

Here's an example of setting up Redux without and with a decorator:

Without a decorator

```

import React from 'react'
import * as actionCreators from './actionCreators'
import { bindActionCreators } from 'redux'
import { connect } from 'react-redux'

function mapStateToProps(state) {
  return { todos: state.todos }
}

```

```

function mapDispatchToProps(dispatch) {
  return { actions: bindActionCreators(actionCreators, dispatch) }
}

class MyApp extends React.Component {
  // ...define your main app here
}

export default connect(mapStateToProps, mapDispatchToProps)(MyApp)

```

Using a decorator

```

import React from 'react'
import * as actionCreators from './actionCreators'
import { bindActionCreators } from 'redux'
import { connect } from 'react-redux'

function mapStateToProps(state) {
  return { todos: state.todos }
}

function mapDispatchToProps(dispatch) {
  return { actions: bindActionCreators(actionCreators, dispatch) }
}

@connect(mapStateToProps, mapDispatchToProps)
export default class MyApp extends React.Component {
  // ...define your main app here
}

```

16Q. What is the difference between React State vs Redux State?

React state is stored locally within a component. When it needs to be shared with other components, it is passed down through props. In practice, this means that the top-most component in your app needing access to a mutable value will hold that value in its state. If it can be mutated by subcomponents, you must pass a callback to handle the change into subcomponents.

When using Redux, state is stored globally in the Redux store. Any component that needs access to a value may subscribe to the store and gain access to that value. Typically, this is done using container components. This centralizes all data but makes it very easy for a component to get the state it needs, without surrounding components knowing of its needs.

17Q. What is the best way to access redux store outside a react component?

To access redux store outside a react component, Redux `connect` function works great for regular React components.

In the examples below shows how to access a JWT token from the Redux store.

Option 1: Export the Store

```
import { createStore } from 'redux'
```

```

import reducer from './reducer'

const store = createStore(reducer)

export default store

```

Here, we are creating the store and exporting it. This will make it available to other files. Here we'll see an `api` file making a call where we need to pass a JWT token to the server:

```

import store from './store'

export function getProtectedThing() {
  // grab current state
  const state = store.getState()

  // get the JWT token out of it
  // (obviously depends on how your store is structured)
  const authToken = state.currentUser.token

  // Pass the token to the server
  return fetch('/user/thing', {
    method: 'GET',
    headers: {
      Authorization: `Bearer ${authToken}`
    }
  }).then(res => res.json())
}

```

Option 2: Pass the Value From a React Component

It's simple to get access to the store inside a React component – no need to pass the store as a prop or import it, just use the `connect()` function from React Redux, and supply a `mapStateToProps()` function that pulls out the data.

```

import React from 'react'
import { connect } from 'react-redux'
import * as api from 'api'

const ItemList = ({ authToken, items }) => {
  return (
    <ul>
      {items.map(item => (
        <li key={item.id}>
          {item.name}
          <button
            onClick={
              () => api.deleteItem(item, authToken)
            }
          >
            DELETE THIS ITEM
          </button>
        </li>
      ))}
    </ul>
  )
}

const mapStateToProps = state => ({
  authToken: state.currentUser && state.currentUser.authToken,
})

```

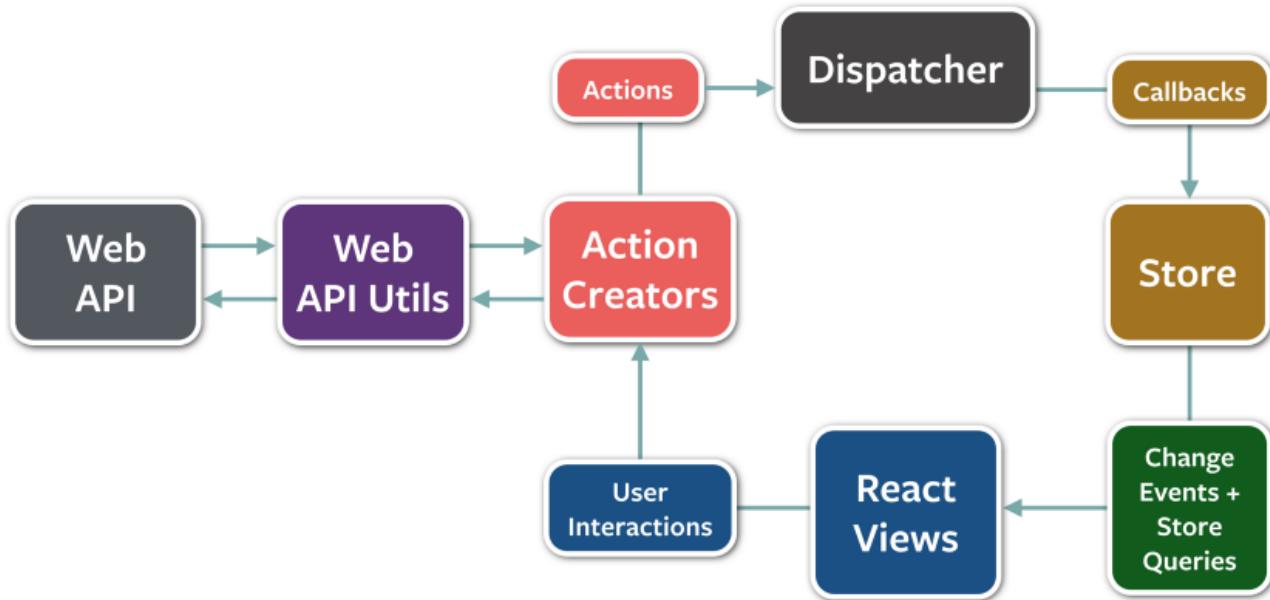
```

    items: state.items
  }

export connect(mapStateToProps)(ItemList)

```

18Q. What are the drawbacks of Redux contrasted with Flux?

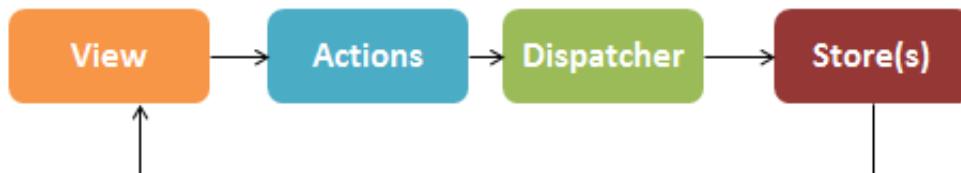


Flux is application architecture or pattern designed, developed and used by Facebook for building user interface or client-side web application. Flux follows unidirectional data flow which supports or empowers composable view of React's components.

Flux Architecture

The Flux architecture is based on the following components:

- **Store/ Stores**: Serves as a container for the app state & logic
- **Action**: Enables data passing to the dispatcher
- **View**: Same as the view in MVC architecture, but in the context of React components
- **Dispatcher**: Coordinates actions & updates to stores



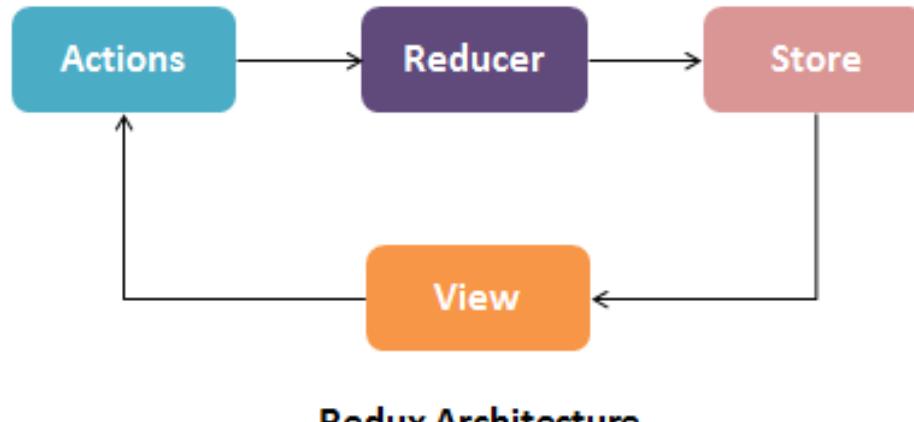
Flux Architecture

In the Flux architecture, when a user clicks on something, the view creates actions. Action can create new data and send it to the dispatcher. The dispatcher then dispatches the action result to the appropriate store. The store updates the state based on the result and sends an update to the view.

Redux Architecture

Redux is a library, which implements the idea of Flux but in quite a different way. Redux architecture introduces new components like:

- **Reducer**: Logic that decides how your data changes exist in pure functions
- **Centralized store**: Holds a state object that denotes the state of the entire app



In Redux architecture, application event is denoted as an Action, which is dispatched to the reducer, the pure function. Then reducer updates the centralized store with new data based on the kind of action it receives. Store creates a new state and sends an update to view. At that time, the view was recreated to reflect the update.

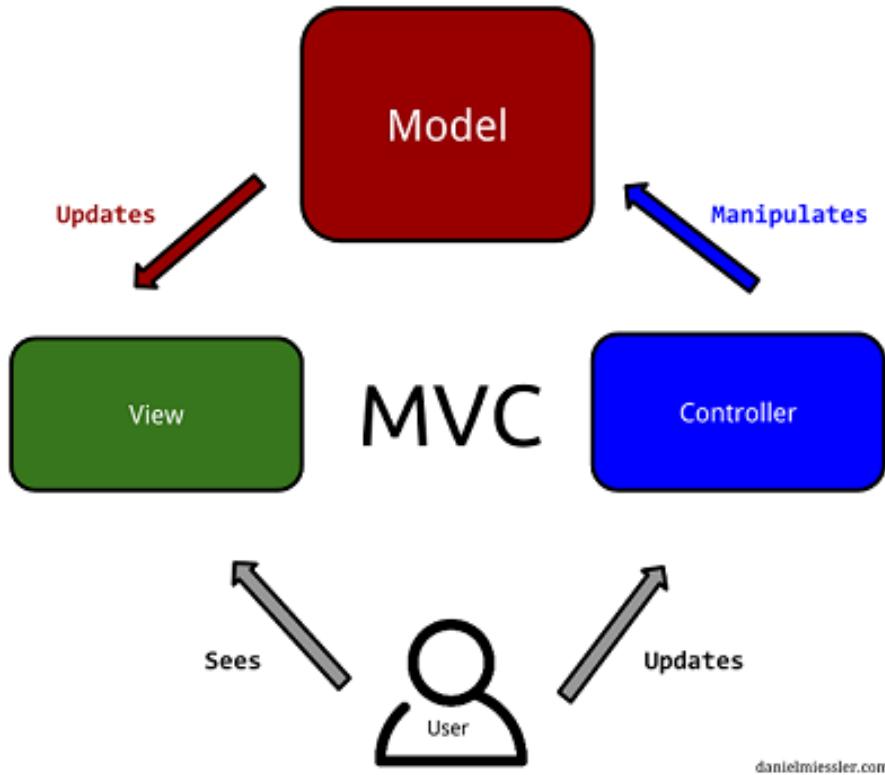
Flux vs Redux

| Flux | Redux |
|--|---|
| Follows the unidirectional flow | Follows the unidirectional flow |
| Includes multiple stores | Includes single store |
| Store handles all logic | Reducer handles all logic |
| Ensures simple debugging with the dispatcher | Single store makes debugging lot easier |

19Q. *Describe Flux vs MVC?*

1. MVC

MVC stands for Model View Controller. It is an architectural pattern used for developing the user interface. It divides the application into three different logical components: the Model, the View, and the Controller.



danielmessier.com

- **Model:** It is responsible for maintaining the behavior and data of an application.
- **View:** It is used to display the model in the user interface.
- **Controller:** It acts as an interface between the Model and the View components. It takes user input, manipulates the data(model) and causes the view to update.

MVC can be interpreted or modified in many ways to fit a particular framework or library. The core ideas of MVC can be formulated as:

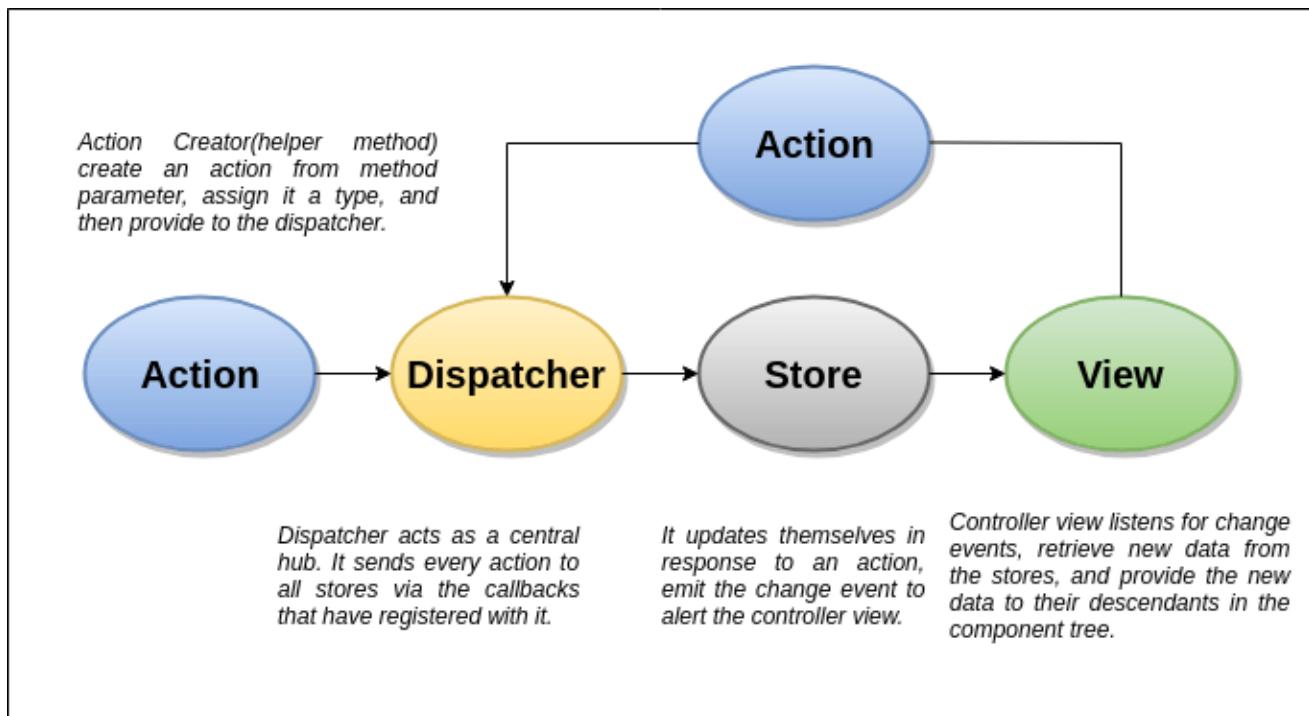
- Separating the presentation from the model: enables implementation of different UIs and better testability
- Separating the controller from the view: most useful with web interfaces and not commonly used in most GUI frameworks

In general, MVC makes no assumptions about whether data flow within an application should be unidirectional or bidirectional. In server Side, MVC is good, but in Client side most of the JS frameworks provide data binding support which let the view can talk with model directly, It shoudn't be, Many times it become hard to debug something as there are scope for a property being changed by many ways.

2. Flux

Flux places unidirectional data flow front and center, by making it a requirement. Here are the four major roles that make up the Flux architecture:

- Actions, which are helper methods that relay information to the dispatcher
- Stores are similar to the models in MVC, except they act as containers for application state and logic for a particular domain within the application
- The Dispatcher receives Actions and acts as the sole registry of callbacks to all stores within an application. It also manages the dependencies between stores
- Views are the same as the view in MVC, except in the context of React and Flux, and also include Controller-Views for change events and retrieve application state from stores as required.



1. All data in the application flow through a central hub called the Dispatcher
2. This data is tracked as actions, which are provided to the dispatcher in an action creator method, often as a result of a user interacting with the view
3. The dispatcher invokes the registered callback, effectively dispatching the action to all stores that have registered with that callback
4. The stores in turn relay that change event to the controller-views to alert them of the change
5. The controller-views listen for events, retrieve data from the appropriate stores as required and re-render themselves and all their children in the component tree accordingly.

MVC Vs. Flux

| MVC | Flux |
|--|---|
| Bidirectional data flow | Unidirectional data flow |
| Data binding is the key | Events or actions are the main players |
| Controllers handle the business logic | Store does all calculations |
| Somewhat synchronous | Can be implemented as completely asynchronous |
| It is hard to debug. | It is easy to debug because it has common initiating point: Dispatcher. |
| Its maintainability is difficult as the project scope goes huge. | Its maintainability is easy and reduces runtime errors. |

20Q. How to add multiple middleware to redux?

The most common use case for middleware is to support asynchronous actions without much boilerplate code or a dependency on a library like RxJS. It does so by letting you dispatch async actions in addition to normal actions.

`applyMiddleware` takes each piece of middleware as a new argument (not an array). It provides a third-party extension point between dispatching an action, and the moment it reaches the reducer. It can be used for logging, crash reporting, talking to an asynchronous API, routing, and more.

```
const createStoreWithMiddleware = applyMiddleware(ReduxThunk,  
logger)(createStore);
```

Example: Custom Logger Middleware

```
import { createStore, applyMiddleware } from 'redux'  
import todos from './reducers'  
  
function logger({ getState }) {  
  return next => action => {  
    console.log('will dispatch', action)  
  
    // Call the next dispatch method in the middleware chain.  
    const returnValue = next(action)  
  
    console.log('state after dispatch', getState())  
  
    // This will likely be the action itself, unless  
    // a middleware further in chain changed it.  
    return returnValue  
  }  
}  
  
const store = createStore(todos, ['Use Redux'], applyMiddleware(logger))  
  
store.dispatch({  
  type: 'ADD_TODO',  
  text: 'Understand the middleware'  
})  
// (These lines will be logged by the middleware:  
// will dispatch: { type: 'ADD_TODO', text: 'Understand the middleware' }  
// state after dispatch: [ 'Use Redux', 'Understand the middleware' ]
```

21Q. How to set initial state in Redux?

1. Initializing State

In Redux, all application state is held in the store; which is an object that holds the complete state tree of your app. There is only one way to change its state and that is by dispatching actions. Actions are objects that consist of a type and a payload property. They are created and dispatched by special functions called action creators.

Example: First creating the Redux store

```
import { createStore } from 'redux'  
  
function todosReducer(state = [], action) {  
  switch (action.type) {  
    case 'ADD_TODO':  
      return state.concat([action.payload])  
    default:  
      return state  
  }  
}
```

```

}

const store = createStore(todosReducer)

```

Next updating the store

```

const ADD_TODO = add_todo; // creates the action type
const newTodo = ["blog on dev.to"];
function todoActionCreator (newTodo) {
  const action = {
    type: ADD_TODO,
    payload: newTodo
  }
  dispatch(action)
}

```

When a store is created, Redux dispatches a dummy action to your reducer to populate the store with the initial state.

2. createStore Pattern

The createStore method can accept an optional preloadedState value as its second argument. In our example, we called `createStore()` without passing this value. When a value is passed to the `preloadedState` it becomes the initial state.

```

const initialState = ["eat", "code", "sleep"];
const store = createStore(todosReducer, initialState)

```

3. Reducer Pattern

Reducers can also specify an initial state value by looking for an incoming state argument that is `undefined`, and returning the value they'd like to use as a default.

```

function todosReducer(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return state.concat([action.payload])
    default:
      return state
  }
}

/**
 * sets initial state to []. But would only take effect if the initial state
 * is undefined,
 * which means it was not set using createStore().
 */

```

In general, `preloadedState` wins over the state specified by the `reducer`. This lets reducers specify initial data that makes sense to them as default arguments, but also allows loading existing data (fully or partially) when you're hydrating the store from some persistent storage or the server.

22Q. Are there any similarities between Redux and RxJS?

Redux:

Predictable state container for JavaScript apps. Redux helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test. On top of that, it provides a great developer experience, such as live code editing combined with a time traveling debugger. However, Redux has one, but very significant problem - it doesn't handle asynchronous operations very well by itself.

RxJS

The Reactive Extensions for JavaScript. RxJS is a library for reactive programming using Observables, to make it easier to compose asynchronous or callback-based code.

Redux belongs to "State Management Library" category of the tech stack, while RxJS can be primarily classified under "Concurrency Frameworks".

Redux

Redux is a tool for managing state throughout the application.

It is usually used as an architecture for UIs.

Redux uses the Reactive paradigm because the Store is reactive. The Store observes actions from a distance, and changes itself.

Example: React, Redux and RxJS

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Subject } from 'rxjs/Subject';

// create our stream as a subject so arbitrary data can be sent on the stream
const action$ = new Subject();

// Initial State
const initState = { name: 'Alex' };

// Redux reducer
const reducer = (state, action) => {
  switch(action.type) {
    case 'NAME_CHANGED':
      return {
        ...state,
        name: action.payload
      };
    default:
      return state;
  }
}

// Reduxification
const store$ = action$
  .startWith(initState)
  .scan(reducer);

// Higher order function to send actions to the stream
const actionDispatcher = (func) => (...args) =>
  action$.next(func(...args));

// Example action function
const changeName = actionDispatcher((payload) => ({
  type: 'NAME_CHANGED',
  payload
}));
```

RxJS

RxJS is a reactive programming library

It is usually used as a tool to accomplish asynchronous tasks in JavaScript.

RxJS also uses the Reactive paradigm, but instead of being an architecture, it gives you basic building blocks, Observables, to accomplish this pattern.

```

// React view component
const App = (props) => {
  const { name } = props;
  return (
    <div>
      <h1>{ name }</h1>
      <button onClick={() => changeName('Alex')}>Alex</button>
      <button onClick={() => changeName('John')}>John</button>
    </div>
  );
}

// subscribe and render the view
const dom = document.getElementById('app');
store$.subscribe((state) =>
  ReactDOM.render(<App {...state} />, dom));

```

Async actions

Let's say we want to do something asynchronous like fetch some information from a rest api all we need to do is send an ajax stream in place of our action payload and then use one of the lodash style stream operators, `flatMap()` to squash the results of the asynchronous operation back onto the `action$` stream.

```

import { isObservable } from './utils';

// Action creator
const actionCreator = (func) => (...args) => {
  const action = func.call(null, ...args);
  action$.next(action);
  if (isObservable(action.payload))
    action$.next(action.payload);
  return action;
};

// method called from button click
const loadUsers = actionCreator(() => {
  return {
    type: 'USERS_LOADING',
    payload: Observable.ajax('/api/users')
      .map(({response}) => map(response, 'username'))
      .map((users) => ({
        type: 'USERS_LOADED',
        payload: users
      }))
  };
});

// Reducer
export default function reducer(state, action) {
  switch (action.type) {
    case 'USERS_LOADING':
      return {
        ...state,
        isLoading: true
      };
  }
}

```

```

        case 'USERS_LOADED':
            return {
                ...state,
                isLoading: false,
                users: action.payload,
            };
        //...
    }
}

// rest of code...

// Wrap input to ensure we only have a stream of observables
const ensureObservable = (action) =>
    isObservable(action)
        ? action
        : Observable.from([action]);

// Using flatMap to squash async streams
const action$ =
    .flatMap(wrapActionToObservable)
    .startWith(initState)
    .scan(reducer);

```

The advantage of swapping the action payload for a stream is so we can send data updates at the start and the end of the async operation

23Q. *What is the purpose of the constants in Redux?*

- It helps keep the naming consistent because all action types are gathered in a single place.
- Sometimes you want to see all existing actions before working on a new feature. It may be that the action you need was already added by somebody on the team, but you didn't know.
- The list of action types that were added, removed, and changed in a Pull Request helps everyone on the team keep track of scope and implementation of new features.
- If you make a typo when importing an action constant, you will get undefined. This is much easier to notice than a typo when you wonder why nothing happens when the action is dispatched.

Example: Constants in Redux can be used into two places, reducers and during actions creation.

```
// actionTypes.js

export const ADD_TODO = 'ADD_TODO'
export const DELETE_TODO = 'DELETE_TODO'
export const EDIT_TODO = 'EDIT_TODO'
export const COMPLETE_TODO = 'COMPLETE_TODO'
export const COMPLETE_ALL = 'COMPLETE_ALL'
export const CLEAR_COMPLETED = 'CLEAR_COMPLETED'
```

And then require it in actions creator file

```
// actions.js

import { ADD_TODO } from './actionTypes'

export function addTodo(text) {
```

```

    return { type: ADD_TODO, text }
}

And in some reducer
import { ADD_TODO } from './actionTypes'

export default (state = [], action) => {
  switch (action.type) {
    case ADD_TODO:
      return [
        ...state,
        {
          text: action.text,
          completed: false
        }
      ]
    default:
      return state
  }
}

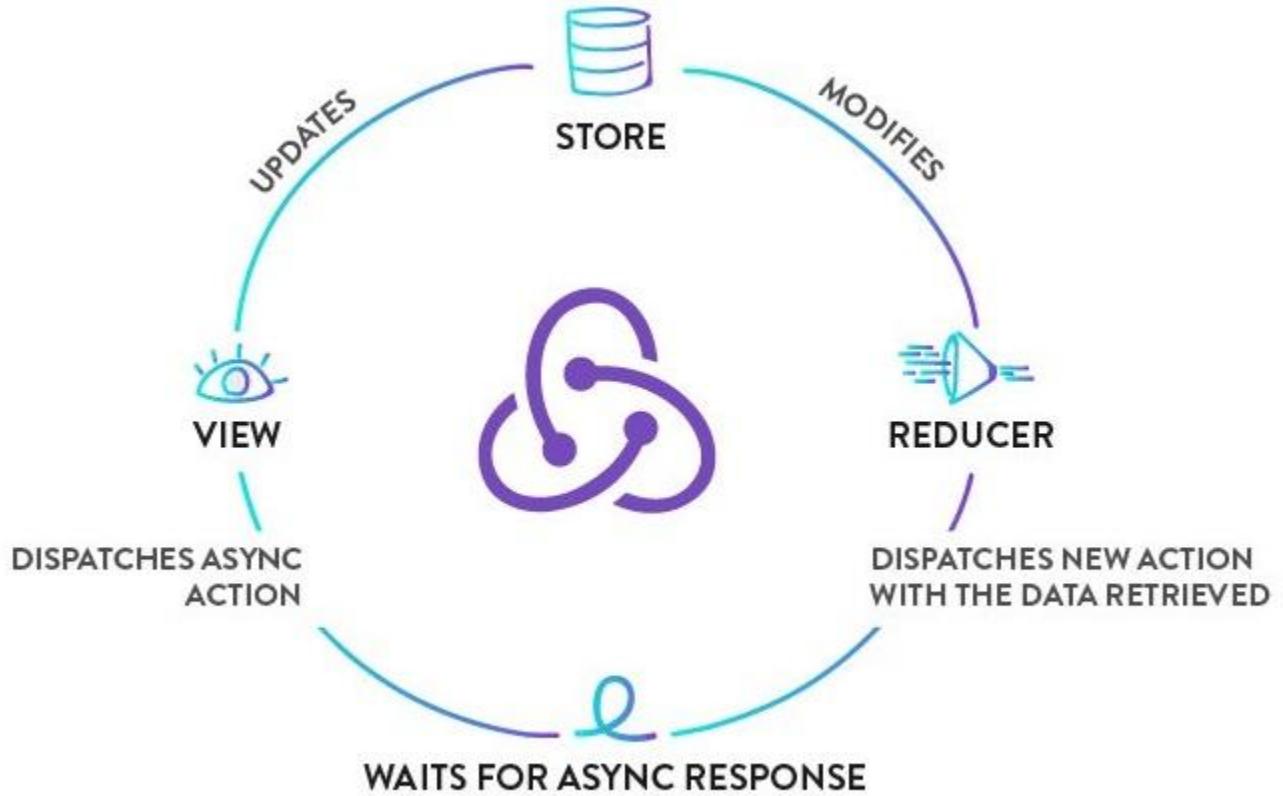
```

It allows to easily find all usages of that constant across the project. It also prevents from introducing silly bugs caused by typos -- in which case, you will get a `ReferenceError` immediately.

24Q. *What are the differences between redux-saga and redux-thunk?*

1. Redux Thunk

Redux Thunk is a middleware that lets you call action creators that return a function instead of an action object. That function receives the store's `dispatch` method, which is then used to dispatch regular synchronous actions inside the body of the function once the asynchronous operations have completed.



```
npm i --save react-redux redux redux-logger redux-saga redux-thunk
```

Thunk is a function which optionally takes some parameters and returns another function, it takes dispatch and getState functions and both of these are supplied by Redux Thunk middleware.

Here is the basic structure of Redux-thunk

```
export const thunkName = parameters => (dispatch, getState) => {
// You can write your application logic here
}
```

Example

```
import axios from "axios"
import GET_LIST_API_URL from "../config"

const fetchList = () => {
  return (dispatch) => {
    axios.get(GET_LIST_API_URL)
      .then((responseData) => {
        dispatch(getList(responseData.list))
      })
      .catch((error) => {
        console.log(error.message)
      })
  }
}

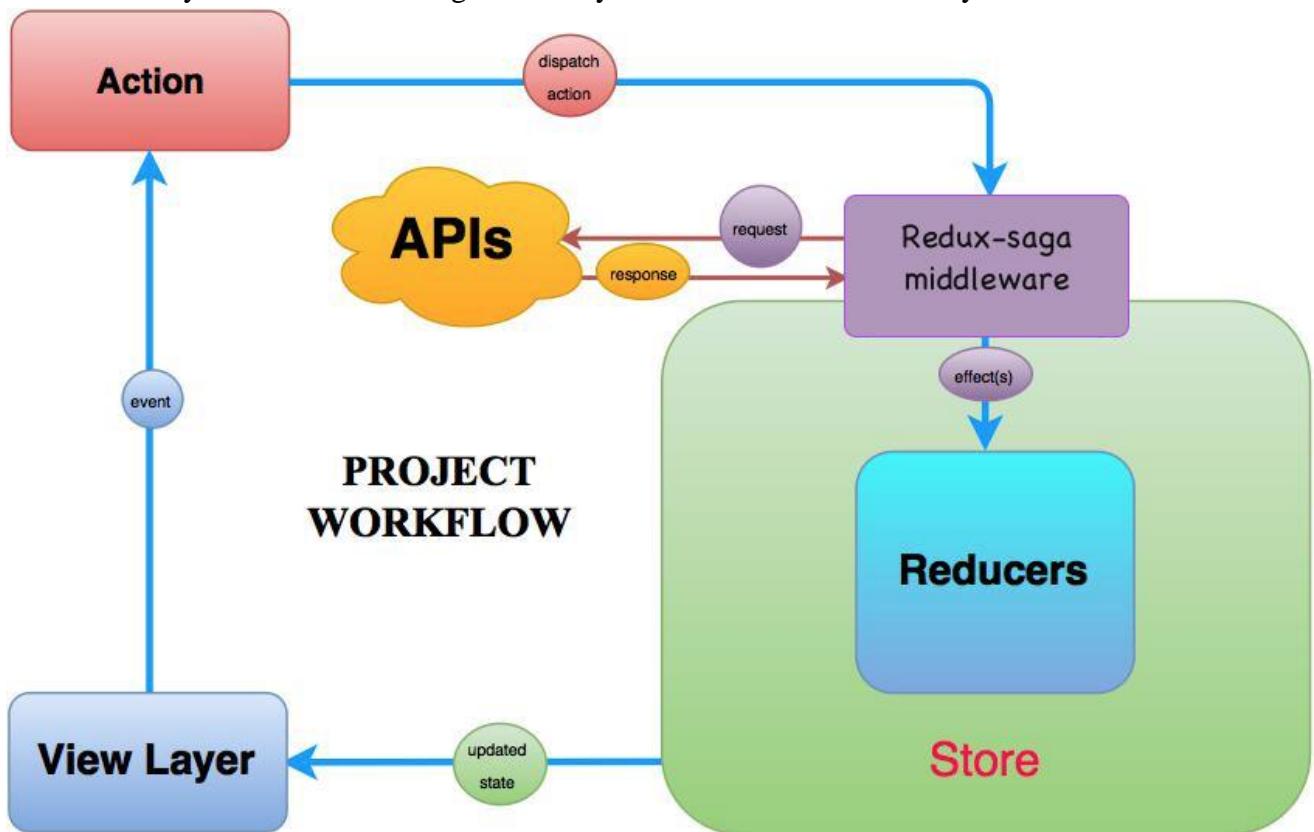

```

```
const getList = (payload) => {
  return {
    type: "GET_LIST",
    payload
  }
}
```

```
    }  
}  
  
export { fetchList }
```

2. Redux Saga

Redux Saga leverages an ES6 feature called `Generators`, allowing us to write asynchronous code that looks synchronous, and is very easy to test. In the saga, we can test our asynchronous flows easily and our actions stay pure. It organized complicated asynchronous actions easily and make them very readable and the saga has many useful tools to deal with asynchronous actions.



Example:

```
import axios from "axios"  
import GET_LIST_API_URL from "../config"  
import {call, put} from "redux-saga/effects"  
  
const fetchList = () => {  
  return axios.get(GET_LIST_API_URL)  
}  
  
function *fetchList () {  
  try {  
    const responseData = yield call(getCharacters)  
    yield put({type: "GET_LIST", payload: responseData.list})  
  } catch (error) {  
    console.log(error.message)  
  }  
}  
  
export { fetchList }
```

Both Redux Thunk and Redux Saga take care of dealing with side effects. In very simple terms, applied to the most common scenario (async functions, specifically AJAX calls) Thunk allows "Promises" to deal with them, Saga uses Generators. Thunk is simple to use and Promises are familiar to many developers, Saga/Generators are more powerful but you will need to learn them. When Promises are just good enough, so is Thunk, when you deal with more complex cases on a regular basis, Saga gives you better tools.

25Q. Explain Redux form with an example?

This is a simple demonstration of how to connect all the standard HTML form elements to redux-form.

For the most part, it is a matter of wrapping each form control in a `<Field>` component, specifying which type of `React.DOM` component you wish to be rendered.

The Field component will provide your input with `onChange`, `onBlur`, `onFocus`, `onDrag`, and `onDrop` props to listen to the events, as well as a **value** prop to make each input a **controlled component**. Notice that the SimpleForm component has no state; in fact, it uses the functional stateless component syntax.

```
// SimpleForm.js

import React from 'react'
import { Field, reduxForm } from 'redux-form'

const SimpleForm = (props) => {
  const { handleSubmit, pristine, reset, submitting } = props
  return (
    <form onSubmit={handleSubmit}>
      <div>
        <label>First Name</label>
        <div>
          <Field name="firstName" component="input" type="text"
placeholder="First Name"/>
        </div>
      </div>
      <div>
        <label>Last Name</label>
        <div>
          <Field name="lastName" component="input" type="text"
placeholder="Last Name"/>
        </div>
      </div>
      <div>
        <label>Email</label>
        <div>
          <Field name="email" component="input" type="email"
placeholder="Email"/>
        </div>
      </div>
      <div>
        <label>Sex</label>
        <div>
          <label><Field name="sex" component="input" type="radio"
value="male"/> Male</label>
```

```

        <label><Field name="sex" component="input" type="radio"
value="female"/> Female</label>
        </div>
    </div>
    <div>
        <label>Favorite Color</label>
        <div>
            <Field name="favoriteColor" component="select">
                <option></option>
                <option value="ff0000">Red</option>
                <option value="00ff00">Green</option>
                <option value="0000ff">Blue</option>
            </Field>
        </div>
    </div>
    <div>
        <label htmlFor="employed">Employed</label>
        <div>
            <Field name="employed" id="employed" component="input"
type="checkbox"/>
        </div>
    </div>
    <div>
        <label>Notes</label>
        <div>
            <Field name="notes" component="textarea"/>
        </div>
    </div>
    <div>
        <button type="submit" disabled={pristine ||
submitting}>Submit</button>
        <button type="button" disabled={pristine || submitting}
onClick={reset}>Clear</button>
    </div>
</form>
)
}

export default reduxForm({
  form: 'simple' // a unique identifier for this form
})(SimpleForm)

```

The screenshot shows a user profile form with the following data:

- First Name:** Pradeep
- Last Name:** Kumar
- Email:** pradeep.vwa@gmail.com
- Sex:** Male (radio button selected)
- Favorite Color:** Red (dropdown menu)
- Employed:** Yes (checkbox checked)
- Notes:** React Redux Form Example !

At the bottom are two buttons: **Submit** and **Clear**.

[]

26Q. How to reset state in redux?

The root reducer would normally delegate handling the action to the reducer generated by `combineReducers()`. However, whenever it receives `USER_LOGOUT` action, it returns the initial state all over again.

```
import { combineReducers } from 'redux';
import AppReducer from './AppReducer';

import UsersReducer from './UsersReducer';
import OrderReducer from './OrderReducer';
import NotificationReducer from './NotificationReducer';
import CommentReducer from './CommentReducer';

/**
 * In order to reset all reducers back to their initial states when user logout,
 * rewrite rootReducer to assign 'undefined' to state when logout
 *
 * If state passed to reducer is 'undefined', then the next state reducer
 * returns
 * will be its initial state instead; since we have assigned it as the
 * default value
 * of reducer's state parameter
 * ex: const Reducer = (state = initialState, action) => { ... }
 *
 * See: https://goo.gl/GSJ98M and combineReducers() source codes for details

```

```

*/
const appReducer = combineReducers({
  /* your app's top-level reducers */
  users: UsersReducer,
  orders: OrderReducer,
  notifications: NotificationReducer,
  comment: CommentReducer,
});

const rootReducer = (state, action) => {
  // when a logout action is dispatched it will reset redux state
  if (action.type === 'USER_LOGGED_OUT') {
    state = undefined;
  }

  return appReducer(state, action);
}

export default rootReducer

```

27Q. Why are Redux state functions called as reducers?

Redux state functions called a reducer because it's the type of function we pass to `Array.prototype.reduce(reducer, ?initialValue)`. Reducers do not just return default values. They always return the accumulation of the state (based on all previous and current actions).

Therefore, they act as a reducer of state. Each time a redux reducer is called, the state is passed in with the action `(state, action)`. This state is then reduced (or accumulated) based on the action, and then the next state is returned. This is one cycle of the classic `fold` or `reduce` function.

28Q. How to make Ajax request in Redux?

There are three most widely used and stable Redux Ajax middleware are:

- Redux Promise Middleware
- Redux Thunk Middleware
- Redux Saga Middleware

1. Redux Promise Middleware

This is the most simple way of doing Ajax calls with Redux. When using Redux Promise, your action creator can return a Promise inside the Action.

```

function.getUserName(userId) {
  return {
    type: "SET_USERNAME",
    payload: fetch(`api/personalDetails/${userId}`)
      .then(response => response.json())
      .then(json => json.userName)
  }
}

```

This middleware automatically dispatches two events when the Ajax call succeeds: `SETUSERNAMEPENDING` and `SETUSERNAMEFULFILLED`. If something fails it dispatches `SETUSERNAMEREJECTED`.

When to use

- You want the simplest thing with minimum overhead
- You prefer convention over configuration
- You have simple Ajax requirements

2. Redux Thunk Middleware

This is the standard way of doing Ajax with Redux. When using Redux Thunk, your action creators returns a function that takes one argument `dispatch`:

```
function getUserName(userId) {
  return dispatch => {
    return fetch(`/api/personalDetails/${userId}`)
      .then(response => response.json())
      .then(json => dispatch({ type: "SET_USERNAME", userName: json.userName }))
  }
}
```

The action creator can call `dispatch` inside `.then` to execute it asynchronously. The action creator can call `dispatch` as many time as it wants.

When to use

- You make many Ajax calls in one action, and need to dispatch many actions
- You require full control of the format of your actions

3. Redux Saga Middleware

This is the most advanced way of doing Ajax with Redux. It uses an ES6 feature called generators. When using Redux Saga you do your Ajax calls in a saga instead of an action creator. This is how a saga looks like:

```
import { call, put, takeEvery } from 'redux-saga/effects'

// call getUserName when action SET_USERNAME is dispatched
function* mySaga() {
  yield takeEvery("SET_USERNAME", getUserName);
}

function* getUserName(action) {
  try {
    const user = yield call(fetch, `/api/personalDetails/${userId}`);
    yield put({type: "SET_USERNAME_SUCCEEDED", user: user});
  } catch (e) {
    yield put({type: "SET_USERNAME_FAILED", message: e.message});
  }
}

export default mySaga
```

Here, sagas listen to actions which you dispatch as regular synchronous actions. In this case, the saga `getUserName` is executed when the action `SET_USERNAME` is dispatched. The `*` next to the function means it's a generator and `yield` is a generator keyword.

When to use

- You need to be able to test the asynchronous flow easily

- You are comfortable working with ES6 Generators
- You value pure functions

29Q. What are the differences between call and put in redux-saga?

Both `call()` and `put()` are effect creator functions. `call()` function is used to create effect description, which instructs middleware to call the promise. `put()` function creates an effect, which instructs middleware to dispatch an action to the store.

Let's take example of how these effects work for fetching particular user data.

```
function* fetchUserSaga(action) {
  // `call` function accepts rest arguments, which will be passed to
  // `api.fetchUser` function.
  // Instructing middleware to call promise, it resolved value will be
  assigned to `userData` variable
  const userData = yield call(api.fetchUser, action.userId)

  // Instructing middleware to dispatch corresponding action.
  yield put({
    type: 'FETCH_USER_SUCCESS',
    userData
  })
}
```

30Q. What is the mental model of redux-saga?

Saga is like a separate thread in your application, that is solely responsible for side effects. `redux-saga` is a redux middleware, which means this thread can be **started**, **paused** and **cancelled** from the main application with normal Redux actions, it has access to the full Redux application state and it can dispatch Redux actions as well.

Example:

```
npm install --save redux-saga
```

Suppose we have a UI to fetch some user data from a remote server when a button is clicked.

```
class UserComponent extends React.Component {
  ...
  onSomeButtonClicked() {
    const { userId, dispatch } = this.props
    dispatch({type: 'USER_FETCH_REQUESTED', payload: {userId}})
  }
  ...
}
```

The Component dispatches a plain Object action to the Store. We'll create a Saga that watches for all `USER_FETCH_REQUESTED` actions and triggers an API call to fetch the user data.

```
// sagas.js

import { call, put, takeEvery, takeLatest } from 'redux-saga/effects'
import Api from '...'

// worker Saga: will be fired on USER_FETCH_REQUESTED actions
function* fetchUser(action) {
  try {
```

```

        const user = yield call(Api.fetchUser, action.payload.userId);
        yield put({type: "USER_FETCH_SUCCEEDED", user: user});
    } catch (e) {
        yield put({type: "USER_FETCH_FAILED", message: e.message});
    }
}

/*
Starts fetchUser on each dispatched `USER_FETCH_REQUESTED` action.
Allows concurrent fetches of user.
*/
function* mySaga() {
    yield takeEvery("USER_FETCH_REQUESTED", fetchUser);
}

/*
Alternatively you may use takeLatest.

Does not allow concurrent fetches of user. If "USER_FETCH_REQUESTED" gets
dispatched while a fetch is already pending, that pending fetch is
cancelled
and only the latest one will be run.
*/
function* mySaga() {
    yield takeLatest("USER_FETCH_REQUESTED", fetchUser);
}

export default mySaga;

```

To run our Saga, we'll have to connect it to the Redux Store using the `redux-saga` middleware.

```
// main.js

import { createStore, applyMiddleware } from 'redux'
import createSagaMiddleware from 'redux-saga'

import reducer from './reducers'
import mySaga from './sagas'

// create the saga middleware
const sagaMiddleware = createSagaMiddleware()
// mount it on the Store
const store = createStore(
    reducer,
    applyMiddleware(sagaMiddleware)
)

// then run the saga
sagaMiddleware.run(mySaga)

// render the application

```

31Q. How Relay is different from Redux?

Redux

Predictable state container for JavaScript apps. Redux helps you write applications that behave consistently, run in different environments (client, server, and native). In redux the application state is located in a single store, each component can access the state, and can also change the state by dispatching actions. Redux doesn't handle data fetching out of the box, though it can be done manually: simply create an action that fetches the data from the server into the store.

Some of the features offered by Redux are:

- Predictable state
- Easy testing
- Works with other view layers besides React

Relay

Created by facebook for react, and also used internally there. Relay is similar to redux in that they both use a single store. The main difference is that relay only manages state originated from the server, and all access to the state is used via GraphQL querys (for reading data) and mutations (for changing data). Relay caches the data for you and optimizes data fetching for you, by fetching only changed data and nothing more. Relay also supports optimistic updates, i.e. changing the state before the server's result arrives.

Relay provides the following key features:

- Build data driven apps
- Declarative style
- Mutate data on the client and server

GraphQL is a web service framework and protocol using declarative and composable queries, and solves problem like over fetching and under fetching, it is believed to be a valid candidate to replace REST.

32Q. When would bindActionCreators be used in react/redux?

`bindActionCreators(actionCreators, dispatch)`: Turns an object whose values are action creators, into an object with the same keys, but with every action creator wrapped into a dispatch call so they may be invoked directly.

When we use Redux with React, react-redux will provide `dispatch()` function and we can call it directly. The only use case for `bindActionCreators()` is when we want to pass some action creators down to a component that isn't aware of Redux, and we don't want to pass `dispatch` or the Redux store to it.

Parameters

1. `actionCreators` (Function or Object): An action creator, or an object whose values are action creators.
2. `dispatch` (Function): A dispatch function available on the Store instance.

Returns

(Function or Object): An object mimicking the original object, but with each function immediately dispatching the action returned by the corresponding action creator. If you passed a function as `actionCreators`, the return value will also be a single function.

Example:

```
// TodoActionCreators.js
```

```
export function addTodo(text) {
  return {
```

```
        type: 'ADD_TODO',
        text
    }
}

export function removeTodo(id) {
    return {
        type: 'REMOVE_TODO',
        id
    }
}
// TodoListContainer.js

import { Component } from 'react'
import { bindActionCreators } from 'redux'
import { connect } from 'react-redux'

import * as TodoActionCreators from './TodoActionCreators'

console.log(TodoActionCreators)
// {
//   addTodo: Function,
//   removeTodo: Function
// }

class TodoListContainer extends Component {
    constructor(props) {
        super(props)

        const { dispatch } = props

        // Here's a good use case for bindActionCreators:
        // You want a child component to be completely unaware of Redux.
        // We create bound versions of these functions now so we can
        // pass them down to our child later.

        this.boundActionCreators = bindActionCreators(TodoActionCreators,
dispatch)
        console.log(this.boundActionCreators)
        // {
        //   addTodo: Function,
        //   removeTodo: Function
        // }
    }

    componentDidMount() {
        // Injected by react-redux:
        let { dispatch } = this.props

        // Note: this won't work:
        // TodoActionCreators.addTodo('Use Redux')

        // You're just calling a function that creates an action.
        // You must dispatch the action, too!
    }
}
```

```

// This will work:
let action = TodoActionCreators.addTodo('Use Redux')
dispatch(action)
}

render() {
  // Injected by react-redux:
  let { todos } = this.props

  return <TodoList todos={todos} {...this.boundActionCreators} />

  // An alternative to bindActionCreators is to pass
  // just the dispatch function down, but then your child component
  // needs to import action creators and know about them.

  // return <TodoList todos={todos} dispatch={dispatch} />
}
}

export default connect(state => ({ todos: state.todos }))(TodoListContainer)

```

33Q. What is *mapStateToProps* and *mapDispatchToProps*?

react-redux package provides 3 functions `Connect`, `mapStateToProps` and `mapDispatchToProps`. `Connect` is a higher order function that takes in both `mapStateToProps` and `mapDispatchToProps` as parameters.

1. Using `MapStateToProps`

In React, `MapStateToProps` pulls in the state of a specific reducer state object from global store and maps it to the props of component. `MapStateToProps` is called everytime your store is updated. You pass in your state a retrieve that specific objects from the reducer.

2. Using `MapDispatchToProps`

`MapDispatchToProps` takes the dispatch functions in component and executes them against the Redux Reducer when that function is fired. `MapDispatchToProps` allows to dispatch state changes to your store.

In a simple term,

mapStateToProps: It connects redux state to props of react component.

mapDispatchToProps: It connects redux actions to react props.

Example:

```

const {createStore} = Redux
const {connect, Provider} = ReactRedux
const initialState = {Collection: ["COW", "COW", "DUCK", "DUCK"]}

function reducer(state=initialState, action) {
  if (action.type === "REVERSE") {
    return Object.assign({}, state, {
      Collection: state.Collection.slice().reverse()
    })
  }
  return state
}

```

```

var store = createStore(reducer)

function mapStateToProps(state) {
  return state
}

var PresentationalComponent = React.createClass({
  render: function() {
    return (
      <div>
        <h2>Store State ( as Props) </h2>
        <pre> {JSON.stringify(this.props.Collection)}</pre>
        <StateChangerUI />
      </div>
    )
  }
})

// State changer UI
var StateChangerUI = React.createClass({
  // Action Dispatch
  handleClick: function() {
    store.dispatch({
      type: 'REVERSE'
    })
  },
  render: function() {
    return (
      <button type="button" className="btn btn-success"
        onClick={this.handleClick}>REVERSE</button>
    )
  }
})

PresentationalComponent = connect(mapStateToProps)(PresentationalComponent)

ReactDOM.render(
  <Provider store={store}>
    <PresentationalComponent />
  </Provider>,
  document.getElementById('App')
)

```

34Q. What is reselect and how it works?

Reselect is a simple library for creating memoized, composable **selector** functions. Reselect selectors can be used to efficiently compute derived data from the Redux store.

Selectors can compute derived data, allowing Redux to store the minimal possible state. Which can be considered as keep the store as minimal as possible. A selector is not recomputed unless one of its arguments change. A memoized selector that recalculates only when that part of the start tree changes which are input arguments to the selector. The value of selector doesn't change when there is no change in other (unrelated) parts of the state tree.

selectors

In our context, selectors are nothing but functions which can compute or retrieve data from the store. We usually fetch the state data using `mapStateToProps()` function like this.

```
const mapStateToProps = (state) => {
  return {
    activeData: getActiveData(state.someData, state.isActive)
  }
}
```

Where `getActiveData()` will be a function which returns all the records from `someData` having the status as `isActive`. The drawback with this function is, whenever any part of the state updates, this function will recalculate this data.

When we use `Reselect` it caches the input arguments to the memoized function. So only when the arguments of the function changes from the previous call, the selector recalculates.

Example:

```
// todo.reducer.js
// ...
import { createSelector } from 'reselect';

const todoSelector = state => state.todo.todos;
const searchTermSelector = state => state.todo.searchTerm;

export const filteredTodos = createSelector(
  [todoSelector, searchTermSelector],
  (todos, searchTerm) => {
    return todos.filter(todo => todo.title.match(new RegExp(searchTerm,
      'i')));
  }
);

// ...
```

We can use the `filteredTodos` selectors to get all the todos if there's no `searchTerm` set in the state, or a filtered list otherwise.

36Q. What are the different ways to dispatch actions in Redux?

Redux is a state container for Javascript apps, mostly used with React. It's based on actions that are dispatched and listened by reducers which modify the state properly.

1. Passing dispatch method to our component

The `dispatch` method is a method of the store object. An action is dispatched to trigger an update to the store.

```
// App.js
import { createStore } from 'redux';
import { MessageSender } from './MessageSender';
import reducer from './reducer';

const store = createStore(reducer);
class App extends React.Component {
  render() {
    <MessageSender store={store} />
  }
};
```

```
// MessageSender.js
import { sendMsg } from './actions';
// ...
this.props.store.dispatch(sendMsg(msg))
// ...
```

2. Using React-Redux to make dumb/smart components

The downside of the above approach is that our React component is aware of the app logic. It's best to separate the logic in our smart component connected to the store from the user interface, i.e., from the dumb component.

From the official docs for `connect()`, we can describe `mapDispatchToProps()` this way: If an object is passed, each function inside it is assumed to be a Redux action creator. An object with the same function names, but with every action creator wrapped into a `dispatch` call so they may be invoked directly, will be merged into the component's props.

```
// MessageSender.container.js
```

```
import { connect } from 'react-redux';
import { sendMsg } from './actions';
import MessageSender from './MessageSender';

const mapDispatchToProps = {
  sendMsg
};

export default connect(null, mapDispatchToProps)(MessageSender);

// MessageSender.js
// ...
this.props.sendMsg(msg);
// ...
```

3. Using the `bindActionCreators()` method

The `bindActionCreators()` method allows us to dispatch actions from any React component that is not connected to the store as `mapDispatchToProps()` in the `connect` function of `react-redux`.

```
// MsgSenderPage.js

import { bindActionCreators } from 'redux';
import { connect } from 'react-redux';
import * as actions from './actions';

class MsgSenderPage extends React.Component {
  constructor(props) {
    super(props);
    const { dispatch } = props;
    this.boundedActions = bindActionCreators(actions, dispatch);
  }

  render() {
    return <MsgSending {...this.boundedActions} />;
  }
}

export default connect()(MsgSenderPage);
```

1. What was the most interesting solution you implemented during your last project?
2. What the latest programming book you've read.
3. How big is/was your team?
4. Have you ever worked in [Agile, Scrum or Kanban](#) environments?
5. Which developers do you know in the Front End community?
6. Are you a team player?
7. What makes you a good developer?
8. What next skill do you want to improve?
9. What values can you bring to a new team?
10. What is the difference between imperative and declarative programming in JS? - [@blog](#)

Common technical interview questions

1. What is REST? - [@docs](#)
2. What is the difference between PUT and PATCH methods in REST? - [@stackoverflow](#)
3. Talk about the differences between [websockets](#), long polling and [SSE](#). - [@stackoverflow](#)
4. What is CORS? - [@blog](#), [@docs](#)
5. List the main things you can do to increase page speed loading? - [@blog](#)
6. Progressive enhancement vs graceful degradation. What is the difference? - [@docs](#)
7. Do you use [Grunt](#), [Gulp](#), [Webpack](#) or [Browserify](#) in your projects?
8. What tools do you know or already use to improve your code?
9. What do you know about "60fps"? How can you achieve it? - [@github](#)
10. What is the difference between layout, painting and compositing? - [@docs](#)
11. What is Web Components? - [@docs](#)
12. What is the difference between sessionStorage, localStorage and cookies?
[@stackoverflow](#)
13. Describe your perfect React/Angular/Vue application stack?

HTML Interview Questions

1. Could you list major HTML5 tags? - [@docs](#)
2. What does an 'optional' closing tag mean? - [@docs](#)
3. When and how to preload resources? - [@blog](#)
4. What is the difference between id and class? - [@blog](#)

CSS Interview Questions

1. What is the difference between 'mobile first' and 'desktop first' - [@blog](#)?
2. Which of [these](#) selectors has the highest specificity. What color will be applied to the paragraph?
3. What does the pseudo-class `:root` refer to?
4. What preprocessor do you use? ([Sass](#) or [Less](#))

Javascript Interview Questions

JS Junior

1. Who is the author of JavaScript Language?
2. What is the best book for learning JavaScript and why? - [@github](#)
3. What is the type of NaN? How to check if a value is NaN?
4. What the reason that `window.window === window` return true? - [@docs](#)
5. What is the outcome of the JavaScript calculation? $1/0 = ?$
6. What is hoisting? [@docs](#)
7. What is the difference between bubbling and capturing? - [@stackoverflow](#)

JS Middle

1. What does `this` refer to? - [@blog](#)
2. What is the JavaScript Event Loop? - [@blog](#), [@youtube](#)
3. What is the Event Delegation? - [@blog](#), [@code](#)
4. What is the difference between `e.target` and `e.currentTarget`? - [@docs](#), [@code](#)
5. What is `Window.postMessage()` and where it can be used? - [@docs](#)
6. Is there any difference between Promises and callbacks? Which is better? - [@blog](#),
7. What is recursion? When is the use of recursion useful in Javascript? - [@blog](#)
8. What do you hear about DRY, KISS, YAGNI? - [@blog](#)
9. What do you know about optional chaining operators? What benefits does it bring? [@docs](#)

JS Senior

1. What patterns do you know and successfully use in JavaScript?
2. What is the problem throttling and debouncing are resolved? What is the core difference between them? - [@blog](#)
3. What is SOLID? [@wiki](#)
4. What is the difference between inheritance and composition? What do you prefer? Why? [@blog](#), [@blog](#)

Javascript Coding Questions

- ▼ Write a `pipefy` function where a string received is returned, but with the "|" character between each character:

```
pipefy();
```

@code

- ▼ Write a currying function that returns a sum of two numbers:

```
sum(1)(2);
```

[currying function](#),

- ▼ Write a factorialfunction without side effect:

```
// Code below must return `true`.
alert(factorial(3) === 6 && factorial(0) === 1);
```

[factorial](#), [side effect](#), @code

- ▼ Which line of the below code will be executed with an error? Why?

```
10 .toString();
(10).toString();
10..toString();
```

- ▼ What is the order of alerts?

```
setTimeout(function () {
  alert('gorilla');
  setTimeout(function () {
    alert('classical inheritance');
  }, 0);
  alert('drumroll');
}, 0);

alert('banana');
```

- ▼ What is the result after code execution: 1, 2 or 3?

```
var x = 1;
var foo = {
  x: 2,
  bar: function () {
    x = 3;
    return this.x;
  },
};
var run = foo.bar;
```

```
alert(run());
```

- ▼ What below code will return: `true` or `false`. What does each part of code return?

```
new String('a') instanceof String && 'b' instanceof String;
```

- ▼ What the result of the following functions call? Is it the same?

```
const a = function (obj, val) {
  obj.val = {
    a: 1,
    b: 2,
  };
  return obj;
};
```

```
const b = function (obj, val) {
  return (obj.val = {
    a: 1,
    b: 2,
  });
};

a({}, 'val');
b({}, 'val');
```

- ▼ What would be the output of this code below?

```
(function () {
  console.log(a, b);
  var a = 1;
  const b = 2;
})();
```

- ▼ Which one of the function expression below would be the best choice for the `prototype-constructor` pattern (a, b, c)? Why?

```
function Man(name) {
  this.name = name;
}
// a
Man.prototype.getName = function () {
  return this.name;
};
// b
Man.prototype.getName = function getName() {
  return this.name;
};
```

```
// c
Man.prototype.getName = () => {
  return this.name;
};
```

- ▼ Implement normalizeCase() function that converts "HELLO worLD" to "Hello world".

```
function normalizeCase(str) {
  // The magic happens here.
}
```

React interview questions

General

1. What is the difference between 'smart and dummy' components?
2. How to create a higher order component?
3. What is the difference between hoc, Renders Props and hooks patterns?
4. Why is it a good practice to render React Components in `<div id="app"></div>` over `<body>` ?
5. What does mean "Isomorphic React Application"? - [@blog](#)
6. What happens when you execute `setState()` in the `render()` method?
7. What is the difference between `useState()` and `useRef()` ?
8. How do lifecycle methods correspond to hooks?
9. Is it okay to call hooks inside loops or conditions?
10. How do you handle errors in React?
11. What is the difference between `useEffect()` and `useLayoutEffect()` ?

Redux

1. What is the difference between Mobx & Redux? - [@blog](#)
2. What do you think about ajax request in reducer?
3. What architecture do you use to structure actions and reducers?
4. What is the difference between reducers and actions?
5. What is an action creator?
6. Is Redux still relevant in 2020? Can React context replace Redux?
7. What is middleware? Can you describe the use case where it can be used?
8. Have you used reselect?

TypeScript interview questions

1. Talk about the differences between public, private, and protected class access modifiers? [@docs](#)
2. What typescript features do you use/know?
3. What is a generic type? [@docs](#)
4. What Typescript utility types do you use/know (Omit, Partial)? [@docs](#)

Security

1. What are the most common types of web attacks? - [@blog](#)

Testing Questions

1. Explain the difference between unit tests and integration tests? - [@stackoverflow](#)
2. Tell about TDD. What advantages or disadvantages of this concept you know? [@docs](#)
3. Which frameworks/platforms do you use for test your code?
4. List unit testing best practices principles. [@slides](#)

General

1. How is Stateless component different from a Stateful component? ↑

The stateless component calculates the internal state of the component but does not have the authority to change state. There is no knowledge about the past, current, or future but receives props from the Stateful component, which are treated as a callback function.

2. What is the difference between state and props? ↑

Both props and state are plain JavaScript objects. While both of them hold information that influences the output of render, they are different in their functionality with respect to component. ie:

- Props get passed to the component similar to function parameters.
- State is managed within the component similar to variables declared within a function.

3. What can you do if the expression contains more than one line? ↑

In such a situation, enclosing the multi-line JSX expression is an option. If you are a first time user, it may seem awkward but later you can understand everything very easily. Many times it becomes necessary to avoid multi-lines to perform the task reliably and for getting the results as expected.

4. What kind of information controls a segment in React? ↑

There are mainly two sorts of information that control a segment: State and Props.

- State: State information that will change, we need to utilize State.
- Props: Props are set by the parent and which are settled all through the lifetime of a part.

5. Explain DOM Diffing in React. ↑

The process of checking the difference between the new VDOM tree and the old VDOM tree is called "diffing". Diffing is accomplished by a heuristic O(n) algorithm. During this process, React will deduce the minimum number of steps needed to

update the real DOM, eliminating unnecessary costly changes. This process is also referred to as reconciliation.

6. What do you understand by the term polling?

The server needs to be monitored to for updates with respect to time. The primary aim in most of the cases is to check whether novel comments are there or not. This process is basically considered as pooling. It checks for updates approximately every 5 seconds. It is possible to change this time period easily. Pooling help keeping an eye on the users and always make sure that no negative information is present on the servers. Actually, it can create issues related to several things and thus pooling is considered.

7. What's the difference between an Element and a Component in React?

Simply put, a React element describes what you want to see on the screen. Not so simply put, a React element is an object representation of some UI.

A React component is a function or a class which optionally accepts input and returns a React element (typically via JSX which gets compiled to a createElement invocation).

8. Why should we not call setState in componentWillUnmount?

You should not call setState() in componentWillUnmount() because the component will never be re-rendered.

9. What is the difference between React Native and React?

React is a JavaScript library, supporting both front end web and being run on the server, for building user interfaces and web applications.

React Native is a mobile framework that compiles to native app components, allowing you to build native mobile applications (iOS, Android, and Windows) in JavaScript that allows you to use React to build your components, and implements React under the hood.

10. What are React Hooks?

Hooks are a new addition in React 16.8. They let you use state and other React

features without writing a class. With Hooks, you can extract stateful logic from a component so it can be tested independently and reused. Hooks allow you to reuse stateful logic without changing your component hierarchy. This makes it easy to share Hooks among many components or with the community.

11. When would you use a Class Component over a Functional Component? ↑

If your component has state or a lifecycle method(s), use a Class component (or Hooks). Otherwise, use a Functional component.

12. How to prevent a function from being called multiple times?



If you use an event handler such as onClick or onScroll and want to prevent the callback from being fired too quickly, then you can limit the rate at which callback is executed. This can be achieved in the below possible ways:

- Throttling: Changes based on a time based frequency. For example, it can be used using `_.throttle` lodash function.
- Debouncing: Publish changes after a period of inactivity. For example, it can be used using `_.debounce` lodash function.
- RequestAnimationFrame throttling: Changes based on `requestAnimationFrame`. For example, it can be used using `raf-schd` lodash function.

13. Describe how events are handled in React. ↑

The event handlers in React will be passed instances of SyntheticEvent to solve cross-browser compatibility issues. As we mentioned earlier, SyntheticEvent is React's cross-browser wrapper around the browser's native event. The synthetic events have the same interface as the native ones but they work identically across all browsers.

However, React doesn't actually attach events to the child nodes themselves. Instead, it uses a single event listener in order to listen to all events at the top level which. Not only is this great for the performance but it also means that React doesn't have to keep track of the event listeners when updating the DOM.

14. How to dispatch the data in-store? ↑

We can dispatch the data to another component which should be based on the action which stores the parent component.

15. What do you understand by “Single source of truth”? ↑

Single source of truth (SSOT) is the practice of structuring information models and associated data schema such that every data element is mastered (or edited) in only one place. Redux uses Store to store the entire state of the application at one location. So all the state of the component is stored in the store and the store itself receives updates. The single state tree makes tracking modifications simpler over time and debugging or inspecting the request.

16. Why is switch keyword used in React Router v4? ↑

Within the Switch component, Route and Redirect components are nested inside. Starting from the top Route/Redirect component in the Switch, to bottom Route/Redirect, each component is evaluated to be true or false based on whether or not the current URL in the browser matches the path/from prop on the Route/Redirect component. Switch is that it will only render the first matched child.

17. How do you tell React to build in Production mode and what will that do? ↑

You set process.env.NODE_ENV to production. When React in production mode, it'll strip out any extra development features like warnings.

18. How do you access imperative API of web components? ↑

Web Components often expose an imperative API to implement its functions. You will need to use a ref to interact with the DOM node directly if you want to access imperative API of a web component. But if you are using third-party Web Components, the best solution is to write a React component that behaves as a wrapper for your Web Component.

19. What is the benefit of strict mode? ↑

It activates additional checks and warnings for its descendants. Note: Strict mode checks are run in development mode only; they do not impact the production build.

It is helpful in the following cases:

- Identifying components with unsafe lifecycle methods.

- Warning about legacy string ref API usage.
- Detecting unexpected side effects.
- Detecting legacy context API.
- Warning about deprecated findDOMNode usage.

20. What is the difference between createElement and cloneElement? ↑

createElement is the thing that JSX gets transpiled to and is the thing that React uses to make React Elements (prostest representations of some UI). cloneElement is utilized as a part of request to clone a component and pass it new props. They nailed the naming on these two.

21. What are the features of ReactJS? ↑

The features of React JS are as follows:

1. React improves SEO performance: React boost the performance of the SEO to higher levels as a search engine faces the problem while reading JavaScript of high loaded applications.
2. React acts as a standard for mobile app development: It provides a transition process as an ideal solution for both mobile and web applications for building rich user interfaces.
3. React makes the process of writing components easier: Using React along with JSX will make you write components and code efficiently and clearly.
4. React increases efficiency: As the React boost the efficiency of components by reusing them. This is the reason why it is considered as an ideal feature of React. It is considered as the most reusable system component.
5. React ensures stable code: It ensures the stability of the code of an application by making use of downward dataflow.

22. What are the rules one needs to follow regarding hooks? ↑

You need to follow two rules in order to use hooks:

- Call Hooks only at the top level of your react functions. i.e, You shouldn't call Hooks inside loops, conditions, or nested functions. This will ensure that Hooks are called in the same order each time a component renders and it preserves the state of Hooks between multiple useState and useEffect calls.
- Call Hooks from React Functions only. i.e, You shouldn't call Hooks from regular JavaScript functions.

23. What are the conditions to safely use the index as a key? ↑

There are three conditions to make sure, it is safe use the index as a key.

- The list and items are static– they are not computed and do not change.
- The items in the list have no ids.
- The list is never reordered or filtered.

24. Explain Presentational segment ↑

A presentational part is a segment which allows you to renders HTML. The segment's capacity is presentational in markup.

25. What is Relay? ↑

Relay is a JavaScript framework for providing a data layer and client-server communication to web applications using the React view layer.

26. Can we make changes inside child components? ↑

Yes, we can make changes inside the child component in Props but not in the case of States.

27. What is route based code splitting? ↑

A good place to start code splitting is with app routes. Break down an application into chunks per route, and then load that chunk when user navigate that route.

Under the hood, webpack takes care of creating chunks and serve chunks to the user on demand.

We have to just create asyncComponent and import the desired component by using dynamic import() function.

28. Explain the purpose of render() in React. ↑

Each React component must have a render() mandatorily. It returns a single React

element which is the representation of the native DOM component. If more than one HTML element needs to be rendered, then they must be grouped together inside one enclosing tag such as `<form>` , `<group>` , `<div>` , etc. This function must be kept pure i.e., it must return the same result each time it is invoked.

29. What is render hijacking? ↑

The concept of render hijacking is the ability to control what a component will output from another component. It actually means that you decorate your component by wrapping it into a Higher-Order component. By wrapping you can inject additional props or make other changes, which can cause changing logic of rendering. It does not actually "ENABLES" hijacking, but by using HOC you make your component behave in different way.

30. What are React Events? ↑

Events are reactions (the library IS called React, right?) that are triggered by specific user actions like clicking on a UI button, hovering a mouse over a UI object, using keyboard commands with the UI, etc.

31. What do you know about Flux? ↑

Basically, Flux is a basic illustration that is helpful in maintaining the unidirectional data stream. It is meant to control construed data unique fragments to make them interface with that data without creating issues. Flux configuration is insipid; it's not specific to React applications, nor is it required to collect a React application. Flux is basically a straightforward idea, however in you have to exhibit a profound comprehension of its usage.

32. What is ReactDOMServer? ↑

The ReactDOMServer object enables you to render components to static markup (typically used on node server). This object is mainly used for server-side rendering (SSR). The following methods can be used in both the server and browser environments:

- `renderToString()`
- `renderToStaticMarkup()`

33. How do the parent and child components exchange information? ↑

This task is generally performed with the help of functions. Actually, there are several functions which are provided to both parent and child components. They simply make use of them through props. Their communication should be accurate and reliable. The need of same can be there anytime and therefore functions are considered for this task. They always make sure that information can be exchanged easily and in an efficient manner among the parent and child components.

34. Where would you put AJAX calls in your React code? ↑

It is possible to use any AJAX library with React, such as Axios, jQuery AJAX, as well as the inbuilt browser `window.fetch`. Data with AJAX calls need to be added to the `componentDidMount()` lifecycle method. By doing this, it is possible to use the `setState()` method for updating component as soon as the data is retrieved.

35. What is the use of Webpack? ↑

Webpack in general is a module bundler, thanks to many plugins it provides although a lot more and it can be used to run tasks, clean build directories, check linting, handle typescript support, increase performance, provide chunking and a lot more.

36. What is the difference between DOM and virtual DOM in React.js? ↑

DOM aka Document Object Model is an abstraction of structured code (HTML). Dom and HTML code are interrelated as the elements of HTML are known as nodes of DOM. It defines a structure where users modify the content present in the structure in any way they want (create, edit, alter, modify etc.). Basically, HTML is a text, DOM is an in-memory representation of this text.

Virtual DOM is a representation of DOM objects like a lightweight copy. It is used and provided for free by React.js

37. Why do we need a Router to React? ↑

We need a Router to React so that we could define the multiple routes whenever the user types a particular URL. This way, the application of a particular router can be made when the URL matches the path defined inside the router.

38. Why did you choose to work with react? ↑

This kind of question is less about rattling off facts related to JSX, Virtual DOMs,

props, or state, and more about explaining to an employer why you have a professional interest in working with React JS. One of the main reasons this question (and your answer) is of interest to an interviewer is because it gives a sense of how you might explain the importance of using React to a non-technical client or stakeholder.

To answer, simply think of what drew you to React JS. It can be something as basic as the fact that React is easy to learn and start with, but allows plenty of room for growth over time (showing your willingness to learn new things and expand your knowledge as you go), or something as practical as the fact that there are so many job opportunities for React developers (showing you keep on top of the industry and are able to adapt as needed).

39. List down the advantages of React Router. ↑

- Just like how React is based on components, in React Router v4, the API is 'All About Components'. A Router can be visualized as a single root component () in which we enclose the specific child routes () .
- No need to manually set History value: In React Router v4, all we need to do is wrap our routes within the component.
- The packages are split: Three packages one each for Web, Native and Core. This supports the compact size of our application. It is easy to switch over based on a similar coding style.

40. What is create-react-app? ↑

create-react-app is the official CLI (Command Line Interface) for React to create React apps with no build configuration. We don't need to install or configure tools like Webpack or Babel. They are preconfigured and hidden so that we can focus on the code. We can install easily just like any other node modules.

41. What are some of the major advantages to using react when building UIs? ↑

Some of the major advantages of using React include:

- Increased application performance via the Virtual DOM model.
- Improved coding efficiency with JSX.
- The ability to reuse components across multiple projects.
- Flexibility and extensibility through add-on tools provided by React's

open source community.

42. Why are you not required to use inheritance? ↑

In React, it is recommended using composition instead of inheritance to reuse code between components. Both Props and composition give you all the flexibility you need to customize a component's look and behavior in an explicit and safe way. Whereas, If you want to reuse non-UI functionality between components, it is suggested to extract it into a separate JavaScript module. Later components import it and use that function, object, or a class, without extending it.

43. How is React different from Angular and VUE? ↑

The core difference between React and Vue is that React lacks any form of “abstraction”. It's very much just straight JavaScript. This brings some of the drawbacks of JS. If you're a JS expert, React will give you more power. But if you are lacking the expertise, Vue will smooth some of the rough patches for you. It's also worth noting that Vue doesn't work with Arrow functions in the same way React does.

VUE.js was launched in 2014 and since then, it has been the most rapidly growing js framework. It is particularly useful for building intuitive interfaces while also being extremely adaptable. VUE is a web application framework that helps in making advanced single page applications.

Angular is a typescript based JavaScript application framework developed by Google, not a collection of libraries and it relies more on HTML than on JS. Despite the slowdown in recent years it's actually used very widely for government and enterprise projects, which depend on a stable, well-established, and consistent ecosystem. It is also known as Super-heroic JavaScript MVW Framework. Its initial purpose was to encounter the challenges of creating single page apps. AngularJS is the oldest version of the Angular framework.

44. What is the second argument that can optionally be passed to setState and what is its purpose? ↑

A callback function which will be invoked when setState has finished and the component is re-rendered.

Since the setState is asynchronous, which is why it takes in a second callback function. With this function, we can do what we want immediately after state has

been updated.

45. What are Higher Order Components(HOC)? ↑

Higher Order Component is an advanced way of reusing the component logic. Basically, it's a pattern that is derived from React's compositional nature. HOC are custom components which wrap another component within it. They can accept any dynamically provided child component but they won't modify or copy any behavior from their input components. You can say that HOC are 'pure' components.

46. What is suspense component? ↑

Suspense is a component that wraps your own custom components. It lets your components communicate to React that they're waiting for some data to load before the component is rendered. It is important to note that Suspense is not a data fetching library like react-async, nor is it a way to manage state like Redux.

47. Is it possible to display props on a parent component? ↑

Yes, it is possible. The best way to perform this task is by using the spread operator. It can also be done with listing the properties but this is a complex process.

48. Name 3 ways to create a component in React and its differences. ↑

There are 3 main ways of creating a component. Extending from the Component class, extending from the PureComponent class or using a stateless function as component. The main difference is that we don't have access to lifecycle methods in a stateless function nor to the state. PureComponent also implements componentShouldUpdate by default and provides a shallow compare for its props and state preventing unnecessary re-renders.

49. Is setState() async? Why? ↑

setState() actions are indeed asynchronous. setState() doesn't immediately mutate this.state. Instead, it creates a pending state transition. Accessing this.state after calling this method can potentially return the existing value. There is no guarantee of synchronous operation of calls to setState and calls may be batched for performance gains.

The reason behind is the way setState alters the state and causes rerendering. Making it synchronous might leave the browser unresponsive. That being said, the

setState calls are asynchronous as well as batched for better UI experience and performance.

50. What is the point of `renderToNodeStream` method? ↑

It is used to render a React element to its initial HTML. Returns a Readable stream that outputs an HTML string. The HTML output by this stream is exactly equal to what `ReactDOMServer.renderToString` would return. You can use this method to generate HTML on the server and send the markup down on the initial request for faster page loads and to allow search engines to crawl your pages for SEO purposes.

51. What is prop drilling and how can you avoid it? ↑

When building a React application, there is often the need for a deeply nested component to use data provided by another component that is much higher in the hierarchy. The simplest approach is to simply pass a prop from each component to the next in the hierarchy from the source component to the deeply nested component. This is called prop drilling. The primary disadvantage of prop drilling is that components that should not otherwise be aware of the data become unnecessarily complicated and are harder to maintain. To avoid prop drilling, a common approach is to use React context. This allows a Provider component that supplies data to be defined, and allows nested components to consume context data via either a Consumer component or a `useContext` hook.

52. What do you understand by “In React, everything is a component.” ↑

Building blocks of the UI of a React application are components. These parts divide the entire UI into tiny parts that are autonomous and reusable. Then it becomes independent of each of these parts without affecting the remainder of the UI.

53. How do you say that state updates are merged? ↑

State update are merged means that when you update only one key in the object state it will not affect the other keys. and key2 would not exist anymore in your state as it was not defined in your update. The merging affects key/value pair which are not included in your update.

54. What is the behavior of uncaught errors in react 16? ↑

In React 16, errors that were not caught by any error boundary will result in unmounting of the whole React component tree. The reason behind this decision is

that it is worse to leave corrupted UI in place than to completely remove it. For example, it is worse for a payments app to display a wrong amount than to render nothing.

55. How does JSX prevent Injection Attacks? ↑

By default, React DOM escapes any values embedded in JSX before rendering them. Thus it ensures that you can never inject anything that's not explicitly written in your application. Everything is converted to a string before being rendered. This helps prevent XSS (cross-site-scripting) attacks.

56. What do you understand by mixin or higher order components in ReactJS? ↑

Higher order components (HOC) is a function that takes component as well as returns a component. It is a modern technique in React that reuses the component logic. However, Higher order components are not a part of React API, per se. These are patterns that emerge from React's compositional nature. In other words, HOC's are functions that loop over and applies a function to every element in an array.

57. What do you understand by Props in React? ↑

Prop is a contraction for Properties in React. These read-only components need to be kept immutable i.e. pure. Throughout the application, props are passed down from the parent components to the child components. In order to maintain the unidirectional data flow, a child component is restricted from sending a prop back to its parent component. This also helps in rendering the dynamically generated data.

58. How is Virtual-DOM more efficient than Dirty checking? ↑

First thing to understand here is that in React, each component has a state which is observable. React knows when to re-render the scene because it is able to observe when this data changes. The observables are significantly faster than the Dirty checking because we don't have to poll the data at a regular interval and check all of the values in the data structure recursively. By comparison, setting a value on the state will signal to a listener that some state has changed. In a situation like that, React can simply listen for change events on the state and queue up re-rendering. Long story short, the virtual DOM is more efficient than the Dirty checking simply because it prevents all the unnecessary re-renders. Re-rendering only occurs when the state changes.

59. Is it mandatory to define constructor for React component? ↑

No, it is not mandatory. i.e, If you don't initialize state and you don't bind methods, you don't need to implement a constructor for your React component.

60. What is the difference between a controlled component and an uncontrolled component? ↑

A large part of React is this idea of having components control and manage their own state. What happens when we throw native HTML form elements (input, select, textarea, etc) into the mix? Should we have React be the “single source of truth” like we’re used to doing with React or should we allow that form data to live in the DOM like we’re used to typically doing with HTML form elements? These two questions are at the heart of controlled vs. uncontrolled components.

A controlled component is a component where React is in control and is the single source of truth for the form data. As you can see below, username doesn’t live in the DOM but instead lives in our component state. Whenever we want to update username, we call setState as we’re used to.

61. What are the rules needs to follow for hooks? ↑

You need to follow two rules inorder to use hooks:

- Call Hooks only at the top level of your react functions. i.e, You shouldn’t call Hooks inside loops, conditions, or nested functions. This will ensure that Hooks are called in the same order each time a component renders and it preserves the state of Hooks between multiple useState and useEffect calls.
- Call Hooks from React Functions only. i.e, You shouldn’t call Hooks from regular JavaScript functions.

62. React has something called a state. What is it and how it is used? ↑

States are the source of data for React components. In other words, they are objects responsible for determining components behavior and rendering. As such, they must be kept as simple as possible. Accessible by means of this.state(), state is mutable and creates dynamic and interactive components.

63. What is JSX? ↑

JSX is a syntax extension to JavaScript and comes with the full power of JavaScript.

JSX produces React “elements”. You can embed any JavaScript expression in JSX by wrapping it in curly braces. After compilation, JSX expressions become regular JavaScript objects. This means that you can use JSX inside of if statements and for loops, assign it to variables, accept it as arguments, and return it from functions. Even though React does not require JSX, it is the recommended way of describing our UI in React app.

64. Explain the use of Redux thunk? ↑

Redux thunk acts as middleware which allows an individual to write action creators that return functions instead of actions. This is also used as a delay function in order to delay dispatch of an action if a certain condition is met. The two store methods `getState()` and `dispatch()` are provided as parameters to the inner function.

65. Is `setState()` is async? Why is `setState()` in React Async instead of Sync? ↑

`setState()` actions are asynchronous and are batched for performance gains.

`setState()` does not immediately mutate `this.state` but creates a pending state transition. Accessing `this.state` after calling this method can potentially return the existing value. There is no guarantee of synchronous operation of calls to `setState` and calls may be batched for performance gains.

This is because `setState` alters the state and causes rerendering. This can be an expensive operation and making it synchronous might leave the browser unresponsive. Thus the `setState` calls are asynchronous as well as batched for better UI experience and performance.

66. What is the difference between async mode and concurrent mode? ↑

Both refer to the same thing. Previously concurrent Mode being referred to as "Async Mode" by React team. The name has been changed to highlight React's ability to perform work on different priority levels. So it avoids the confusion from other approaches to Async Rendering.

67. What does `shouldComponentUpdate` do and why is it important? ↑

What `shouldComponentUpdate` does is it's a lifecycle method that allows us to opt

out of this reconciliation process for certain components (and their child components). Why would we ever want to do this? As mentioned above, “The end goal of reconciliation is to, in the most efficient way possible, update the UI based on new state”. If we know that a certain section of our UI isn’t going to change, there’s no reason to have React go through the trouble of trying to figure out if it should. By returning false from `shouldComponentUpdate`, React will assume that the current component, and all its child components, will stay the same as they currently are.

68. Explain React Decorators ↑

Decorators in React help you take an existing Class component, or function of a Class component, and modify it, thereby allowing you to add extra capabilities, without having to mess with the existing codebase. Modification can be overriding the existing function completely, or just adding extra logic to it. In essence, decorators take one function, and return another function after spicing it up. React and decorators can go hand-in-hand due to presence of Higher Order Components , and Higher Order Functions.

69. What are the different phases of React component’s lifecycle? ↑

There are three different phases of React component’s lifecycle: Initial Rendering Phase: This is the phase when the component is about to start its life journey and make its way to the DOM. Updating Phase: Once the component gets added to the DOM, it can potentially update and re-render only when a prop or state change occurs. That happens only in this phase. Unmounting Phase: This is the final phase of a component’s life cycle in which the component is destroyed and removed from the DOM.

70. What is the purpose of eslint plugin for hooks? ↑

The ESLint plugin enforces rules of Hooks to avoid bugs. It assumes that any function starting with “use” and a capital letter right after it is a Hook. In particular, the rule enforces that, - Calls to Hooks are either inside a PascalCase function (assumed to be a component) or another `useSomething` function (assumed to be a custom Hook). - Hooks are called in the same order on every render.

71. How would you debug an issue in react code? What debugging tools have you used? ↑

Debugging as a crucial part of the development process. Before you start your next React JS job interview, make sure you have experience with the following industry standard debugging tools (and can explain how you'd use them):

- Linters (eslint, jslint)
- Debuggers (React Developer Tools)

72. What are the lifecycle methods of ReactJS? ↑

- `componentWillMount`: Executed before rendering and is used for App level configuration in your root component.
- `componentDidMount`: Executed after first rendering and here all AJAX requests, DOM or state updates, and set up eventListeners should occur.
- `componentWillReceiveProps`: Executed when particular prop updates to trigger state transitions.
- `shouldComponentUpdate`: Determines if the component will be updated or not. By default it returns true. If you are sure that the component doesn't need to render after state or props are updated, you can return false value. It is a great place to improve performance as it allows you to prevent a rerender if component receives new prop.
- `componentWillUpdate`: Executed before re-rendering the component when there are pros & state changes confirmed by `shouldComponentUpdate` which returns true.
- `componentDidUpdate`: Mostly it is used to update the DOM in response to prop or state changes.
- `componentWillUnmount`: It will be used to cancel any outgoing network requests, or remove all event listeners associated with the component.

73. How is ReactJs different from AngularJS? ↑

The first difference between both of them is their code dependency. ReactJS depends less on the code whereas AngularJS needs a lot of coding to be done. The packaging on React is quite strong as compared to the AngularJS. Another difference is React is equipped with Virtual Dom while the Angular has a Regular DOM. ReactJS is all about the components whereas AngularJS focus mainly on the Models, View as well as on Controllers. AngularJS was developed by Google while the ReactJS is the outcome of facebook. These are some of the common differences between the two.

74. What would be two of the most significant drawbacks of React? ↑

- Integrating React with the MVC framework like Rails requires complex configuration.
- React requires the users to have knowledge about the integration of user interface into MVC framework.

75. What are synthetic events in React? ↑

Synthetic events are the objects which act as a cross-browser wrapper around the browser's native event. They combine the behavior of different browsers into one API. This is done to make sure that the events show consistent properties across different browsers.

76. Can you force a React component to rerender without calling setState? ↑

In your component, you can call this.forceUpdate() to force a rerender.

77. What is arrow function in React? How is it used? ↑

Arrow functions are more of brief syntax for writing the function expression. They are also called 'fat arrow' (=>) the functions. These functions allow to bind the context of the components properly since in ES6 auto binding is not available by default. Arrow functions are mostly useful while working with the higher order functions.

78. Mention the key benefits of Flux? ↑

Applications that are built on Flux have components that can simply be tested. By simply updating the store, developers are able to manage and test any react component. It cut down the overall risk of data affection. All the applications are highly scalable and suffer no compatibility issues.

79. Why are fragments better than container divs? ↑

- It's a tiny bit faster and has less memory usage (no need to create an extra DOM node). This only has a real benefit on very large and/or deep trees, but application performance often suffers from death by a thousand cuts. This is one cut less.
- Some CSS mechanisms like Flexbox and CSS Grid have a special

parent-child relationship, and adding divs in the middle makes it hard to keep the desired layout while extracting logical components.

- The DOM inspector is less cluttered.

80. What are refs in React? ↑

Refs is the short hand for References in React. It is an attribute which helps to store a reference to a particular React element or component, which will be returned by the components render configuration function. It is used to return references to a particular element or component returned by render(). They come in handy when we need DOM measurements or to add methods to the components.

81. Is it ref argument available for all functions or class components? ↑

Regular function or class components don't receive the ref argument, and ref is not available in props either. The second ref argument only exists when you define a component with React.forwardRef call.

82. In ReactJS, why there is a need to capitalize on the components? ↑

It is necessary because components are not the DOM element but they are constructors. If they are not capitalized, they can cause various issues and can confuse developers with several elements. At the same time, the problem of integration of some elements and commands can be there.

83. What are the benefits of using typescript with reactjs? ↑

Below are some of the benefits of using typescript with ReactJS:

- It is possible to use latest JavaScript features.
- Use of interfaces for complex type definitions.
- IDEs such as VS Code was made for TypeScript.
- Avoid bugs with the ease of readability and Validation.

84. How would you structure a React application? ↑

This is an open question with many possible answers. The basic structure is usually module or feature based. We usually differentiate between UI and logic. There are many approaches to structure UI components with the most popular being atomic design. Data and business heavy applications use a more domain driven approach.

The ideal combination for larger applications is having the domain logic separate and having the UI logic in an atomic structure. All this can be combined in features which are rendered on pages.

85. What is the use of a super keyword in React? ↑

The super keyword helps you to access and call functions on an object's parent.

86. Why are String Refs considered legacy? ↑

If you worked with React before, you might be familiar with an older API where the ref attribute is a string, like `ref={'textInput'}`, and the DOM node is accessed as `this.refs.textInput`. We advise against it because string refs have below issues, and are considered legacy. String refs were removed in React v16.

- They force React to keep track of currently executing component. This is problematic because it makes react module stateful, and thus causes weird errors when react module is duplicated in the bundle.
- They are not composable — if a library puts a ref on the passed child, the user can't put another ref on it. Callback refs are perfectly composable.
- They don't work with static analysis like Flow. Flow can't guess the magic that framework does to make the string ref appear on `this.refs`, as well as its type (which could be different). Callback refs are friendlier to static analysis.
- It doesn't work as most people would expect with the "render callback" pattern

87. When should you use the top-class elements for the function element? ↑

If your element does a stage or lifetime cycle, we should use top-class elements.

88. What is the methods order when component is re-rendered?



An update can be caused by changes to props or state. The below methods are called in the following order when a component is being re-rendered.

- `static getDerivedStateFromProps()`
- `shouldComponentUpdate()`
- `render()`

- `getSnapshotBeforeUpdate()`
- `componentDidUpdate()`

89. Explain the Virtual DOM and its working. ↑

A virtual DOM is a lightweight JS object. It is simply a copy of the real DOM. A virtual DOM is a node tree that lists various elements, their attributes, and content as objects and their properties.

The `render()` function in React is responsible for creating a node tree from the React components. This tree is then updated in response to the mutations resulting in the data model due to various actions made by the user or the system.

Virtual DOM operates in three simple steps:

- Step 1 – The entire UI is re-rendered in Virtual DOM representation as soon as there are some underlying data changes.
- Step 2 – Now, the difference between the previous DOM representation and the new one (resulted from underlying data changes) is calculated.
- Step 3 – After the calculations are successfully carried out, the real DOM is updated in line with only the things that actually underwent changes.

90. Is it possible to nest JSX elements into other JSX elements? ↑

It is possible. The process is quite similar to that of nesting the HTML elements. However, there are certain things that are different in this. You must be familiar with the source and destination elements to perform this task simply.

Advanced

1. How does React know when to re-render App component if we handle window resizing in useWindowSize? ↑

When you call `setSize` inside the custom hooks, React knows that this hook is used in `App` component and will re-render it.

2. Explain Flexbox and its benefits ↑

Flexbox is a layout-ing method which solves many previous problems with handling layouts in CSS. It eliminates the need for different GRID libraries and using floats to

position blocks on the site. It has a very intuitive api and gives a lot more control and flexibility.

3. Why do we need a key property? Give an example when a bad key causes an error. ↑

There are classic diffing algorithms with $O(n^3)$ time complexity, which might be used for creating a tree of React elements. But it means for displaying 1000 elements would require one billion comparisons.

Instead, React implements a heuristic $O(n)$ algorithm with an assumption that the developer can hint at which child elements may be stable across different renders with a keyprop.

What about a bad key? Well, an index might be a very bad key if you decide to make your children removable. Check out this demo. Try to type something in the second input and then remove the first one. But you still can see the value in the second one, why so?

Because your keys are unstable. After removal, your third child with a key equals to 3, now has a key equals to 2. It's not the same element for React now. And it will match it to the wrong DOM element, which previously had a key equals to 2 (which keeps the value we typed in a second input).

4. React unit tests vs integration tests for components. ↑

It's worth mentioning both Enzyme and react-testing-library.

React testing library provides a clean and simple API which focuses on testing applications “as a user would”. This means an API returns HTML Elements rather than React Components with shallow rendering in Enzyme. It's a nice tool for writing integrational tests.

Enzyme is still a valid tool, it provides a more sophisticated API which gives you access to component's props and internal state. It makes sense to create unit tests for components.

5. What is windowing technique? ↑

Windowing is a technique that only renders a small subset of your rows at any given

time, and can dramatically reduce the time it takes to re-render the components as well as the number of DOM nodes created. If your application renders long lists of data then this technique is recommended. Both react-window and react-virtualized are popular windowing libraries which provides several reusable components for displaying lists, grids, and tabular data.

6. Explain the positives and negatives of shallow rendering components in tests. ↑

Positives:

- It is faster to shallow render a component than to fully render it. When a React project contains a large number of components, this performance difference can have a significant impact on the total time taken for unit tests to execute.
- Shallow rendering prevents testing outside the boundaries of the component being tested—a best practice of unit testing.

Negatives:

- Shallow rendering is less similar to real-world usage of a component as part of an application, so it may not catch certain problems. Take the example of a `<House />` component that renders a `<LivingRoom />` component. Within a real application, if the `<LivingRoom />` component is broken and throws an error, then `<House />` would fail to render. However, if the unit tests of `<House />` only use shallow rendering, then this issue will not be identified unless `<LivingRoom />` is also covered with unit tests.

7. What are the problems of using render props with pure components? ↑

If you create a function inside a render method, it negates the purpose of pure component. Because the shallow prop comparison will always return false for new props, and each render in this case will generate a new value for the render prop. You can solve this issue by defining the render function as instance method.

8. Do you know what the reconciliation algorithm is? ↑

It is the algorithm responsible for figuring out what changed between re-renders and how to update the actual DOM. It is basically a differencing algorithm. The latest addition of improvements on the core algorithm is called React Fiber.

9. How to prevent components from re-rendering? ↑

- `shouldComponentUpdate()` — returns ‘true’ by default. You can override if you know which props have to trigger an update.
- `PureComponents` — The difference between them is that `React.Component` doesn’t implement `shouldComponentUpdate` method but `React.PureComponent` implements it with a shallow prop and state comparison.
- `React.memo` — The same as the previous one but it works with functional components.

10. How would you optimise the performance of a React application? ↑

The most expensive task in a React app is the update of the DOM. The basic optimisation is to reduce how many times a component re-renders. This can be achieved by using `componentShouldUpdate`, using `PureComponent` or memoization libraries like `reselect`. Reducing the size of the final JS file also helps improving performance and we can use dynamic imports and chunks for this.

11. What are the drawbacks of MVW pattern? ↑

- DOM manipulation is very expensive which causes applications to behave slow and inefficient.
- Due to circular dependencies, a complicated model was created around models and views.
- Lot of data changes happens for collaborative applications(like Google Docs).
- No way to do undo (travel back in time) easily without adding so much extra code.

Redux

1. What are the advantages of formik over redux form library? ↑

Below are the main reasons to recommend `formik` over `redux form library`:

- The form state is inherently short-term and local, so tracking it in Redux (or any kind of Flux library) is unnecessary.
- `Redux-Form` calls your entire top-level Redux reducer multiple times ON

EVERY SINGLE KEYSTROKE. This way it increases input latency for large apps.

- Redux-Form is 22.5 kB minified gzipped whereas Formik is 12.7 kB

2. Can Redux only be used with React? ↑

Redux can be used as a data store for any UI layer. The most common usage is with React and React Native, but there are bindings available for Angular, Angular 2, Vue, Mithril, and more. Redux simply provides a subscription mechanism which can be used by any other code.

3. How Relay is different from Redux? ↑

Relay is similar to Redux in that they both use a single store. The main difference is that relay only manages state originated from the server, and all access to the state is used via GraphQL queries (for reading data) and mutations (for changing data). Relay caches the data for you and optimizes data fetching for you, by fetching only changed data and nothing more.

4. What are selectors? Why would you use reselect or a memoization library? ↑

Selectors are functions which accept the state and return a portion of it while applying calculations, transformations, mappings, filtering etc. This way the logic of how to retrieve data for a specific view is encapsulated in a selector. Since many of the mentioned operations are expensive, when calling the selector again without state change, you want to skip the expensive operations as they will return the same results and hence the usage of reselect. Reselect will return the results from the Cashe in case arguments didn't change.

5. What is the mental model of redux-saga? ↑

Saga is like a separate thread in your application, that's solely responsible for side effects. redux-saga is a redux middleware, which means this thread can be started, paused and cancelled from the main application with normal Redux actions, it has access to the full Redux application state and it can dispatch Redux actions as well.

6. What is an action in Redux? ↑

It is a function which returns an action object. The action-type and the action data are always stored in the action object. Actions can send data between the Store and the software application. All information retrieved by the Store is produced by the

actions.

7. Why are Redux state functions called reducers? ↑

Reducers always return the accumulation of the state (based on all previous and current actions). Therefore, they act as a reducer of state. Each time a Redux reducer is called, the state and action are passed as parameters. This state is then reduced (or accumulated) based on the action, and then the next state is returned. You could reduce a collection of actions and an initial state (of the store) on which to perform these actions to get the resulting final state.

8. What are the core principles of Redux? ↑

Redux follows three fundamental principles:

- Single source of truth: The state of your whole application is stored in an object tree within a single store. The single state tree makes it easier to keep track of changes over time and debug or inspect the application.
- State is read-only: The only way to change the state is to emit an action, an object describing what happened. This ensures that neither the views nor the network callbacks will ever write directly to the state.
- Changes are made with pure functions: To specify how the state tree is transformed by actions, you write reducers. Reducers are just pure functions that take the previous state and an action as parameters, and return the next state.

9. What are the downsides of Redux compared to Flux? ↑

- You will need to learn to avoid mutations: Flux is un-opinionated about mutating data, but Redux doesn't like mutations and many packages complementary to -- Redux assumes you never mutate the state. You can enforce this with dev-only packages like redux-immutable-state-invariant, Immutable.js, or instructing your team to write non-mutating code.
- You're going to have to carefully pick your packages: While Flux explicitly doesn't try to solve problems such as undo/redo, persistence, or forms, Redux has extension points such as middleware and store enhancers, and it has spawned a rich ecosystem.
- There is no nice Flow integration yet: Flux currently lets you do very impressive static type checks which Redux doesn't support yet.

| No. | Questions |
|-----|--|
| 325 | How do you make sure that user remains authenticated on page refresh while using Context API State Management? |
| 326 | What are the benefits of new JSX transform? |
| 327 | How does new JSX transform different from old transform? |
| 328 | How do you get redux scaffolding using create-react-app? |
| 329 | What are React Server components? |

Core React

1. What is React?

React is an **open-source frontend JavaScript library** which is used for building user interfaces especially for single page applications. It is used for handling view layer for web and mobile apps. React was created by [Jordan Walke](#), a software engineer working for Facebook. React was first deployed on Facebook's News Feed in 2011 and on Instagram in 2012.

 [Back to Top](#)

2. What are the major features of React?

The major features of React are:

- It uses **VirtualDOM** instead of **RealDOM** considering that **RealDOM** manipulations are expensive.
- Supports **server-side rendering**.
- Follows **Unidirectional** data flow or data binding.
- Uses **reusable/composable** UI components to develop the view.

 [Back to Top](#)

3. What is JSX?

JSX is a XML-like syntax extension to ECMAScript (the acronym stands for *JavaScript XML*). Basically it just provides syntactic sugar for the `React.createElement()` function, giving us expressiveness of JavaScript along with HTML like template syntax.

In the example below text inside `<h1>` tag is returned as JavaScript function to the render function.

```
class App extends React.Component {
```

```
render() {
  return(
    <div>
      <h1>'Welcome to React world!'</h1>
    </div>
  )
}
```

[↑ Back to Top](#)

4. What is the difference between Element and Component?

An *Element* is a plain object describing what you want to appear on the screen in terms of the DOM nodes or other components. *Elements* can contain other *Elements* in their props. Creating a React element is cheap. Once an element is created, it is never mutated.

The object representation of React Element would be as follows:

```
const element = React.createElement(
  'div',
  {id: 'login-btn'},
  'Login'
)
```

The above `React.createElement()` function returns an object:

```
{
  type: 'div',
  props: {
    children: 'Login',
    id: 'login-btn'
  }
}
```

And finally it renders to the DOM using `ReactDOM.render()`:

```
<div id='login-btn'>Login</div>
```

Whereas a **component** can be declared in several different ways. It can be a class with a `render()` method. Alternatively, in simple cases, it can be defined as a function. In either case, it takes props as an input, and returns a JSX tree as the output:

```
const Button = ({ onLogin }) =>
  <div id={'login-btn'} onClick={onLogin}>Login</div>
```

Then JSX gets transpiled to a `React.createElement()` function tree:

```
const Button = ({ onLogin }) => React.createElement(
  'div',
  { id: 'login-btn', onClick: onLogin },
  'Login'
)
```

[↑ Back to Top](#)

5. How to create components in React?

There are two possible ways to create a component.

- i. **Function Components:** This is the simplest way to create a component. Those are pure JavaScript functions that accept props object as first parameter and return React elements:

```
function Greeting({ message }) {
  return <h1>{`Hello, ${message}`}</h1>

}
```

- ii. **Class Components:** You can also use ES6 class to define a component. The above function component can be written as:

```
class Greeting extends React.Component {
  render() {
    return <h1>{`Hello, ${this.props.message}`}</h1>
  }
}
```

[↑ Back to Top](#)

6. When to use a Class Component over a Function Component?

If the component needs *state* or *lifecycle methods* then use class component otherwise use function component. However, from *React 16.8 with the addition of Hooks, you could use state, lifecycle methods and other features that were only available in class component right in your function component.*

[↑ Back to Top](#)

7. What are Pure Components?

`React.PureComponent` is exactly the same as `React.Component` except that it handles the `shouldComponentUpdate()` method for you. When props or state changes, `PureComponent` will do a shallow comparison on both props and state. `Component` on the other hand won't compare current props and state to next out of the box. Thus, the component will re-render by default whenever `shouldComponentUpdate` is called.

[↑ Back to Top](#)

8. What is state in React?

`State` of a component is an object that holds some information that may change over the lifetime of the component. We should always try to make our state as simple as possible and minimize the number of stateful components.

Let's create an user component with message state,

```
class User extends React.Component {
  constructor(props) {
    super(props)

    this.state = {
      message: 'Welcome to React world'
    }
  }

  render() {
    return (
      <div>
        <h1>{this.state.message}</h1>
      </div>
    )
  }
}
```



state is used for internal communication inside a Component

State is similar to props, but it is private and fully controlled by the component. i.e, It is not accessible to any component other than the one that owns and sets it.

[↑ Back to Top](#)

9. What are props in React?

Props are inputs to components. They are single values or objects containing a set of values that are passed to components on creation using a naming convention similar to HTML-tag attributes. They are data passed down from a parent component to a child component.

The primary purpose of props in React is to provide following component functionality:

- i. Pass custom data to your component.
- ii. Trigger state changes.
- iii. Use via `this.props.reactProp` inside component's `render()` method.

For example, let us create an element with `reactProp` property:

```
<Element reactProp={'1'} />
```

This `reactProp` (or whatever you came up with) name then becomes a property attached to React's native props object which originally already exists on all components created using React library.

```
props.reactProp
```

[↑ Back to Top](#)

10. What is the difference between state and props?

Both *props* and *state* are plain JavaScript objects. While both of them hold information that influences the output of render, they are different in their functionality with respect to component. Props get passed to the component similar to function parameters whereas state is managed within the component similar to variables declared within a function.

 [Back to Top](#)

11. Why should we not update the state directly?

If you try to update state directly then it won't re-render the component.

```
//Wrong  
this.state.message = 'Hello world'
```

Instead use `setState()` method. It schedules an update to a component's state object. When state changes, the component responds by re-rendering.

```
//Correct  
this.setState({ message: 'Hello World' })
```

Note: You can directly assign to the state object either in *constructor* or using latest javascript's class field declaration syntax.

 [Back to Top](#)

12. What is the purpose of callback function as an argument of `setState()` ?

The callback function is invoked when `setState` finished and the component gets rendered. Since `setState()` is **asynchronous** the callback function is used for any post action.

Note: It is recommended to use lifecycle method rather than this callback function.

```
setState({ name: 'John' }), () => console.log('The name has updated and component r
```

 [Back to Top](#)

13. What is the difference between HTML and React event handling?

Below are some of the main differences between HTML and React event handling,

i. In HTML, the event name should be in *lowercase*:

```
<button onclick='activateLasers()'>
```

Whereas in React it follows *camelCase* convention:

```
<button onClick={activateLasers}>
```

ii. In HTML, you can return `false` to prevent default behavior:

```
<a href='#' onclick='console.log("The link was clicked."); return false;' />
```

Whereas in React you must call `preventDefault()` explicitly:

```
function handleClick(event) {
  event.preventDefault()
  console.log('The link was clicked.')
}
```

iii. In HTML, you need to invoke the function by appending `()` Whereas in react you should not append `()` with the function name. (refer "activateLasers" function in the first point for example)

 [Back to Top](#)

14. How to bind methods or event handlers in JSX callbacks?

There are 3 possible ways to achieve this:

i. **Binding in Constructor:** In JavaScript classes, the methods are not bound by default. The same thing applies for React event handlers defined as class methods. Normally we bind them in constructor.

```
class Component extends React.Component {
  constructor(props) {
    super(props)
    this.handleClick = this.handleClick.bind(this)
  }

  handleClick() {
    // ...
  }
}
```

ii. **Public class fields syntax:** If you don't like to use bind approach then *public class fields syntax* can be used to correctly bind callbacks.

```
handleClick = () => {
  console.log('this is:', this)
}
```

```
<button onClick={this.handleClick}>
  {'Click me'}
</button>
```

iii. **Arrow functions in callbacks:** You can use *arrow functions* directly in the callbacks.

```
<button onClick={(event) => this.handleClick(event)}>
  {'Click me'}
</button>
```

Note: If the callback is passed as prop to child components, those components might do an extra re-rendering. In those cases, it is preferred to go with `.bind()` or *public class fields syntax* approach considering performance.

 [Back to Top](#)

15. How to pass a parameter to an event handler or callback?

You can use an *arrow function* to wrap around an *event handler* and pass parameters:

```
<button onClick={() => this.handleClick(id)} />
```

This is an equivalent to calling `.bind`:

```
<button onClick={this.handleClick.bind(this, id)} />
```

Apart from these two approaches, you can also pass arguments to a function which is defined as arrow function

```
<button onClick={this.handleClick(id)} />
handleClick = (id) => () => {
  console.log("Hello, your ticket number is", id)
};
```

 [Back to Top](#)

16. What are synthetic events in React?

`SyntheticEvent` is a cross-browser wrapper around the browser's native event. Its API is same as the browser's native event, including `stopPropagation()` and `preventDefault()`, except the events work identically across all browsers.

[↑ Back to Top](#)

17. What are inline conditional expressions?

You can use either *if statements* or *ternary expressions* which are available from JS to conditionally render expressions. Apart from these approaches, you can also embed any expressions in JSX by wrapping them in curly braces and then followed by JS logical operator `&&`.

```
<h1>Hello!</h1>
{
  messages.length > 0 && !isLoggedIn?
    <h2>
      You have {messages.length} unread messages.
    </h2>
    :
    <h2>
      You don't have unread messages.
    </h2>
}
```

[↑ Back to Top](#)

18. What is "key" prop and what is the benefit of using it in arrays of elements?

A `key` is a special string attribute you **should** include when creating arrays of elements. `Key` prop helps React identify which items have changed, are added, or are removed.

Most often we use ID from our data as `key`:

```
const todoItems = todos.map((todo) =>
  <li key={todo.id}>
    {todo.text}
  </li>
)
```

When you don't have stable IDs for rendered items, you may use the item `index` as a `key` as a last resort:

```
const todoItems = todos.map((todo, index) =>
```

```
<li key={index}>
  {todo.text}
</li>
)
```

Note:

- i. Using *indexes* for *keys* is **not recommended** if the order of items may change. This can negatively impact performance and may cause issues with component state.
- ii. If you extract list item as separate component then apply *keys* on list component instead of *li* tag.
- iii. There will be a warning message in the console if the *key* prop is not present on list items.

 [Back to Top](#)

19. What is the use of refs?

The *ref* is used to return a reference to the element. They *should be avoided* in most cases, however, they can be useful when you need a direct access to the DOM element or an instance of a component.

 [Back to Top](#)

20. How to create refs?

There are two approaches

- i. This is a recently added approach. *Refs* are created using `React.createRef()` method and attached to React elements via the *ref* attribute. In order to use *refs* throughout the component, just assign the *ref* to the instance property within constructor.

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props)
    this.myRef = React.createRef()
  }
  render() {
    return <div ref={this.myRef} />
  }
}
```

- ii. You can also use ref callbacks approach regardless of React version. For example, the search bar component's input element accessed as follows,

```
class SearchBar extends Component {
```

```

constructor(props) {
  super(props);
  this.txtSearch = null;
  this.state = { term: '' };
  this.setInputSearchRef = e => {
    this.txtSearch = e;
  }
}
onInputChange(event) {
  this.setState({ term: this.txtSearch.value });
}
render() {
  return (
    <input
      value={this.state.term}
      onChange={this.onInputChange.bind(this)}
      ref={this.setInputSearchRef} />
  );
}
}

```

You can also use *refs* in function components using **closures**. **Note:** You can also use inline ref callbacks even though it is not a recommended approach

 [Back to Top](#)

21. What are forward refs?

Ref forwarding is a feature that lets some components take a *ref* they receive, and pass it further down to a child.

```

const ButtonElement = React.forwardRef((props, ref) => (
  <button ref={ref} className="CustomButton">
    {props.children}
  </button>
));

// Create ref to the DOM button:
const ref = React.createRef();
<ButtonElement ref={ref}>'Forward Ref'</ButtonElement>

```

 [Back to Top](#)

22. Which is preferred option with in callback refs and findDOMNode()?

It is preferred to use *callback refs* over `findDOMNode()` API. Because `findDOMNode()` prevents certain improvements in React in the future.

The **legacy** approach of using `findDOMNode` :

```

class MyComponent extends Component {
  componentDidMount() {
    findDOMNode(this).scrollIntoView()
  }

  render() {
    return <div />
  }
}

```

The recommended approach is:

```

class MyComponent extends Component {
  constructor(props){
    super(props);
    this.node = createRef();
  }
  componentDidMount() {
    this.node.current.scrollIntoView();
  }

  render() {
    return <div ref={this.node} />
  }
}

```

 [Back to Top](#)

23. Why are String Refs legacy?

If you worked with React before, you might be familiar with an older API where the `ref` attribute is a string, like `ref={'textInput'}`, and the DOM node is accessed as `this.refs.textInput`. We advise against it because *string refs have below issues*, and are considered legacy. String refs were **removed in React v16**.

- i. They *force React to keep track of currently executing component*. This is problematic because it makes react module stateful, and thus causes weird errors when react module is duplicated in the bundle.
- ii. They are *not composable* — if a library puts a ref on the passed child, the user can't put another ref on it. Callback refs are perfectly composable.
- iii. They *don't work with static analysis* like Flow. Flow can't guess the magic that framework does to make the string ref appear on `this.refs`, as well as its type (which could be different). Callback refs are friendlier to static analysis.
- iv. It doesn't work as most people would expect with the "render callback" pattern (e.g.)

```
class MyComponent extends Component {
```

```

    renderRow = (index) => {
      // This won't work. Ref will get attached to DataTable rather than MyComp
      return <input ref={'input-' + index} />;
    }

    // This would work though! Callback refs are awesome.
    return <input ref={input => this['input-' + index] = input} />;
  }

  render() {
    return <DataTable data={this.props.data} renderRow={this.renderRow} />
  }
}

```

[↑ Back to Top](#)

24. What is Virtual DOM?

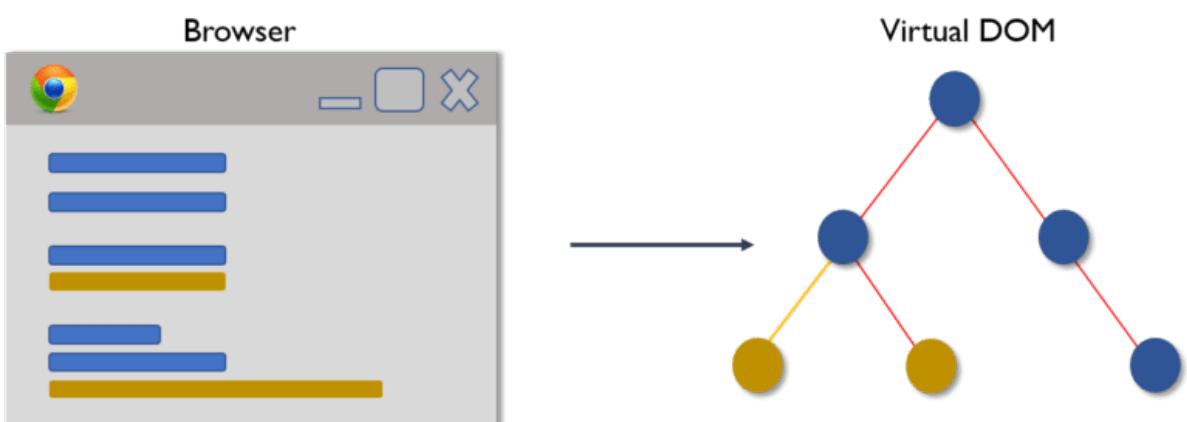
The *Virtual DOM* (VDOM) is an in-memory representation of *Real DOM*. The representation of a UI is kept in memory and synced with the "real" DOM. It's a step that happens between the render function being called and the displaying of elements on the screen. This entire process is called *reconciliation*.

[↑ Back to Top](#)

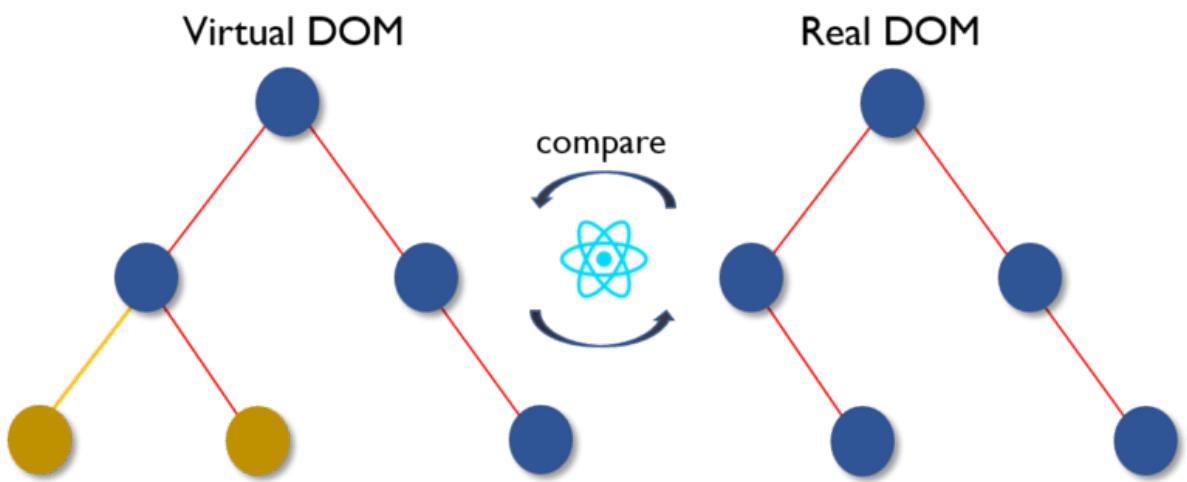
25. How Virtual DOM works?

The *Virtual DOM* works in three simple steps.

- Whenever any underlying data changes, the entire UI is re-rendered in Virtual DOM representation.

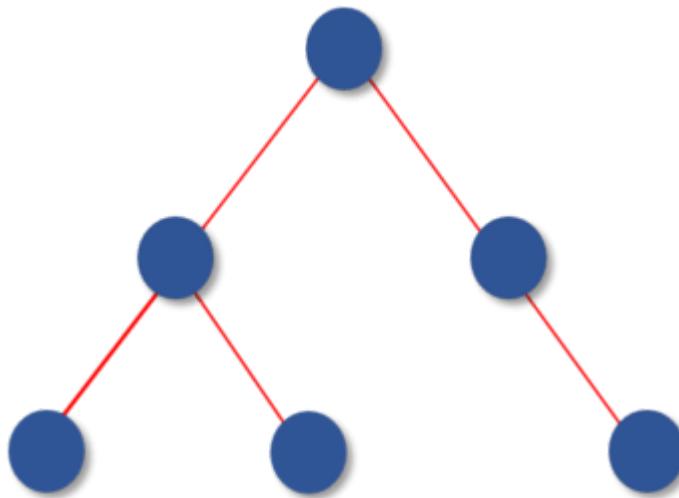


- Then the difference between the previous DOM representation and the new one is calculated.



iii. Once the calculations are done, the real DOM will be updated with only the things that have actually changed.

Real DOM (updated)



[↑ Back to Top](#)

26. What is the difference between Shadow DOM and Virtual DOM?

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.

[↑ Back to Top](#)

27. What is React Fiber?

Fiber is the new *reconciliation* engine or reimplementation of core algorithm in React v16. The goal of React Fiber is to increase its suitability for areas like animation, layout, gestures, ability to pause, abort, or reuse work and assign priority to different types of updates; and new concurrency primitives.

[↑ Back to Top](#)

28. What is the main goal of React Fiber?

The goal of *React Fiber* is to increase its suitability for areas like animation, layout, and gestures. Its headline feature is **incremental rendering**: the ability to split rendering work into chunks and spread it out over multiple frames.

[↑ Back to Top](#)

29. What are controlled components?

A component that controls the input elements within the forms on subsequent user input is called **Controlled Component**, i.e, every state mutation will have an associated handler function.

For example, to write all the names in uppercase letters, we use handleChange as below,

```
handleChange(event) {
  this.setState({value: event.target.value.toUpperCase()})
}
```

[↑ Back to Top](#)

30. What are uncontrolled components?

The **Uncontrolled Components** are the ones that store their own state internally, and you query the DOM using a ref to find its current value when you need it. This is a bit more like traditional HTML.

In the below UserProfile component, the `name` input is accessed using ref.

```
class UserProfile extends React.Component {
  constructor(props) {
    super(props)
    this.handleSubmit = this.handleSubmit.bind(this)
    this.input = React.createRef()
  }

  handleSubmit(event) {
    alert('A name was submitted: ' + this.input.current.value)
    event.preventDefault()
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          {'Name:'}
          <input type="text" ref={this.input} />
        </label>
      </form>
    )
  }
}
```

```
        <input type="submit" value="Submit" />
    </form>
)
}
}
```

In most cases, it's recommended to use controlled components to implement forms.

[↑ Back to Top](#)

31. What is the difference between createElement and cloneElement?

JSX elements will be transpiled to `React.createElement()` functions to create React elements which are going to be used for the object representation of UI. Whereas `cloneElement` is used to clone an element and pass it new props.

[↑ Back to Top](#)

32. What is Lifting State Up in React?

When several components need to share the same changing data then it is recommended to *lift the shared state up* to their closest common ancestor. That means if two child components share the same data from its parent, then move the state to parent instead of maintaining local state in both of the child components.

[↑ Back to Top](#)

33. What are the different phases of component lifecycle?

The component lifecycle has three distinct lifecycle phases:

- i. **Mounting:** The component is ready to mount in the browser DOM. This phase covers initialization from `constructor()`, `getDerivedStateFromProps()`, `render()`, and `componentDidMount()` lifecycle methods.
- ii. **Updating:** In this phase, the component gets updated in two ways, sending the new props and updating the state either from `setState()` or `forceUpdate()`. This phase covers `getDerivedStateFromProps()`, `shouldComponentUpdate()`, `render()`, `getSnapshotBeforeUpdate()` and `componentDidUpdate()` lifecycle methods.
- iii. **Unmounting:** In this last phase, the component is no longer needed and gets unmounted from the browser DOM. This phase includes `componentWillUnmount()` lifecycle method.

It's worth mentioning that React internally has a concept of phases when applying changes to the DOM. They are separated as follows

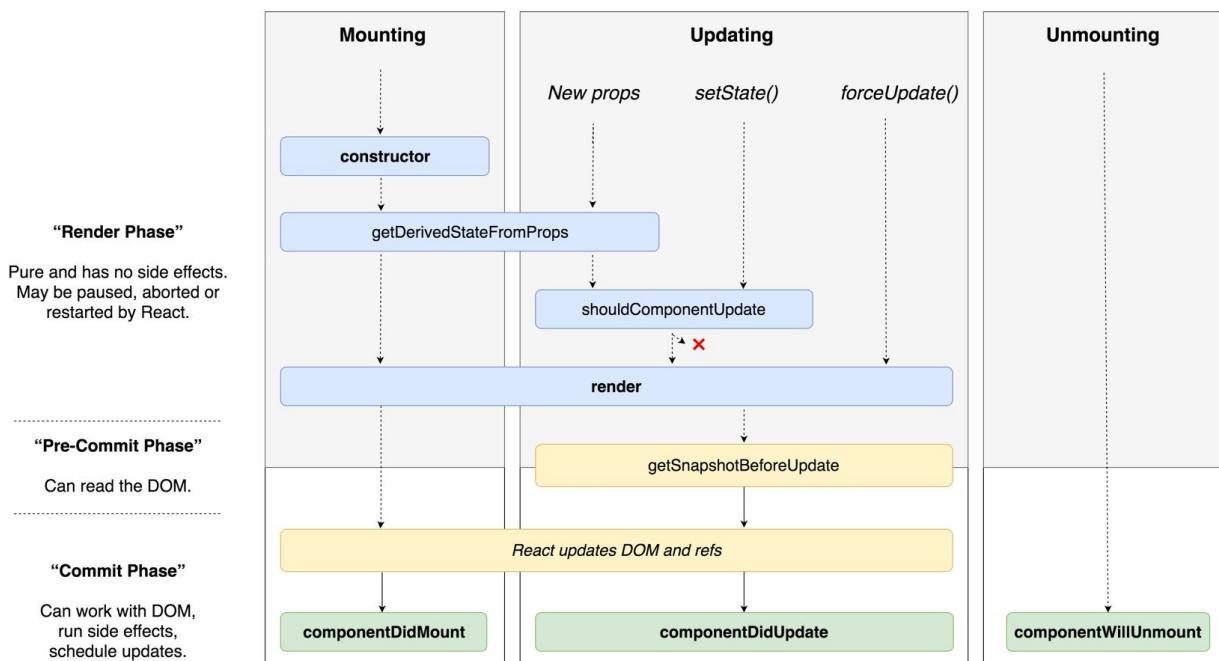
i.

Render The component will render without any side-effects. This applies for Pure components and in this phase, React can pause, abort, or restart the render.

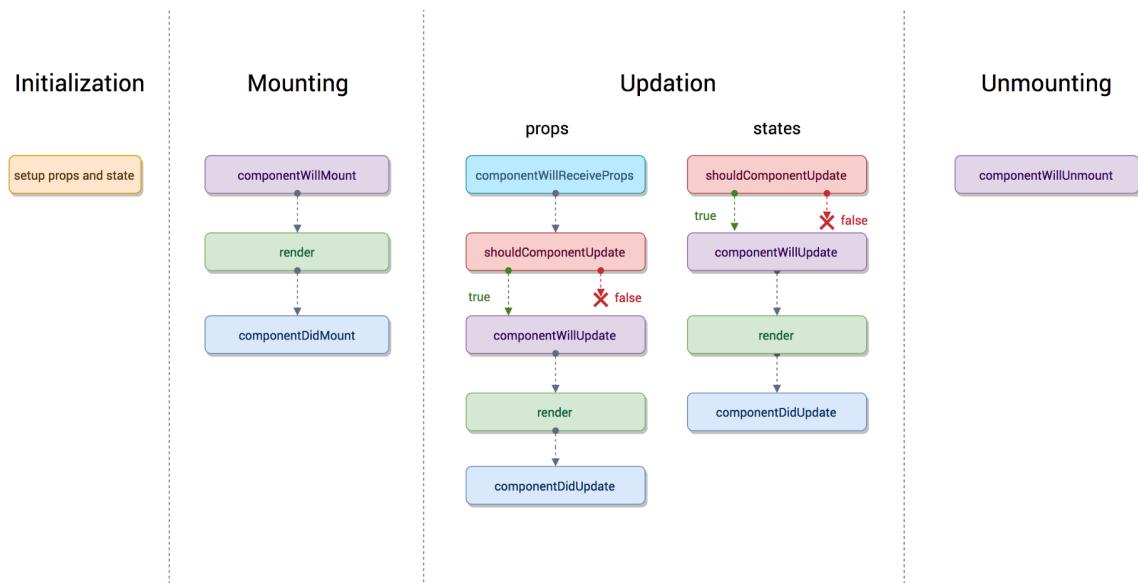
ii. **Pre-commit** Before the component actually applies the changes to the DOM, there is a moment that allows React to read from the DOM through the `getSnapshotBeforeUpdate()`.

iii. **Commit** React works with the DOM and executes the final lifecycles respectively `componentDidMount()` for mounting, `componentDidUpdate()` for updating, and `componentWillUnmount()` for unmounting.

React 16.3+ Phases (or an [interactive version](#))



Before React 16.3



34. What are the lifecycle methods of React?

Before React 16.3

- **componentWillMount:** Executed before rendering and is used for App level configuration in your root component.
- **componentDidMount:** Executed after first rendering and here all AJAX requests, DOM or state updates, and set up event listeners should occur.
- **componentWillReceiveProps:** Executed when particular prop updates to trigger state transitions.
- **shouldComponentUpdate:** Determines if the component will be updated or not. By default it returns `true`. If you are sure that the component doesn't need to render after state or props are updated, you can return false value. It is a great place to improve performance as it allows you to prevent a re-render if component receives new prop.
- **componentWillUpdate:** Executed before re-rendering the component when there are props & state changes confirmed by `shouldComponentUpdate()` which returns true.
- **componentDidUpdate:** Mostly it is used to update the DOM in response to prop or state changes.
- **componentWillUnmount:** It will be used to cancel any outgoing network requests, or remove all event listeners associated with the component.

React 16.3+

- **getDerivedStateFromProps:** Invoked right before calling `render()` and is invoked on every render. This exists for rare use cases where you need derived state. Worth reading [if you need derived state](#).
- **componentDidMount:** Executed after first rendering and where all AJAX requests, DOM or state updates, and set up event listeners should occur.
- **shouldComponentUpdate:** Determines if the component will be updated or not. By default it returns `true`. If you are sure that the component doesn't need to render after state or props are updated, you can return false value. It is a great place to improve performance as it allows you to prevent a re-render if component receives new prop.
- **getSnapshotBeforeUpdate:** Executed right before rendered output is committed to the DOM. Any value returned by this will be passed into `componentDidUpdate()`. This is useful to capture information from the DOM i.e. scroll position.
- **componentDidUpdate:** Mostly it is used to update the DOM in response to prop or state changes. This will not fire if `shouldComponentUpdate()` returns `false`.
- **componentWillUnmount** It will be used to cancel any outgoing network requests, or remove all event listeners associated with the component.

35. What are Higher-Order Components?

A *higher-order component (HOC)* is a function that takes a component and returns a new component. Basically, it's a pattern that is derived from React's compositional nature.

We call them **pure components** because they can accept any dynamically provided child component but they won't modify or copy any behavior from their input components.

```
const EnhancedComponent = higherOrderComponent(WrappedComponent)
```

HOC can be used for many use cases:

- i. Code reuse, logic and bootstrap abstraction.
- ii. Render hijacking.
- iii. State abstraction and manipulation.
- iv. Props manipulation.

 [Back to Top](#)

36. How to create props proxy for HOC component?

You can add/edit props passed to the component using *props proxy* pattern like this:

```
function HOC(WrappedComponent) {  
  return class Test extends Component {  
    render() {  
      const newProps = {  
        title: 'New Header',  
        footer: false,  
        showFeatureX: false,  
        showFeatureY: true  
      }  
  
      return <WrappedComponent {...this.props} {...newProps} />  
    }  
  }  
}
```

 [Back to Top](#)

37. What is context?

Context provides a way to pass data through the component tree without having to pass props down manually at every level.

For example, authenticated user, locale preference, UI theme need to be accessed in the application by many components.

```
const {Provider, Consumer} = React.createContext(defaultValue)
```

[↑ Back to Top](#)

38. What is children prop?

Children is a prop (`this.props.children`) that allow you to pass components as data to other components, just like any other prop you use. Component tree put between component's opening and closing tag will be passed to that component as `children` prop.

There are a number of methods available in the React API to work with this prop. These include `React.Children.map` , `React.Children.forEach` , `React.Children.count` , `React.Children.only` , `React.Children.toArray` .

A simple usage of children prop looks as below,

```
const MyDiv = React.createClass({
  render: function() {
    return <div>{this.props.children}</div>
  }
})

ReactDOM.render(
  <MyDiv>
    <span>'Hello'</span>
    <span>'World'</span>
  </MyDiv>,
  node
)
```

[↑ Back to Top](#)

39. How to write comments in React?

The comments in React/JSX are similar to JavaScript Multiline comments but are wrapped in curly braces.

Single-line comments:

```
<div>
  /* Single-line comments(In vanilla JavaScript, the single-line comments are rep
   `Welcome ${user}, let's play React` }
</div>
```

Multi-line comments:

```
<div>
  /* Multi-line comments for more than
   one line */
  `Welcome ${user}, let's play React`
</div>
```

[↑ Back to Top](#)

40. What is the purpose of using super constructor with props argument?

A child class constructor cannot make use of `this` reference until `super()` method has been called. The same applies for ES6 sub-classes as well. The main reason of passing `props` parameter to `super()` call is to access `this.props` in your child constructors.

Passing props:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props)

    console.log(this.props) // prints { name: 'John', age: 42 }
  }
}
```

Not passing props:

```
class MyComponent extends React.Component {
  constructor(props) {
    super()

    console.log(this.props) // prints undefined

    // but props parameter is still available
    console.log(props) // prints { name: 'John', age: 42 }
  }

  render() {
    // no difference outside constructor
    console.log(this.props) // prints { name: 'John', age: 42 }
  }
}
```

The above code snippets reveals that `this.props` is different only within the constructor. It would be the same outside the constructor.

[↑ Back to Top](#)

41. What is reconciliation?

When a component's props or state change, React decides whether an actual DOM update is necessary by comparing the newly returned element with the previously rendered one. When they are not equal, React will update the DOM. This process is called *reconciliation*.

[↑ Back to Top](#)

42. How to set state with a dynamic key name?

If you are using ES6 or the Babel transpiler to transform your JSX code then you can accomplish this with *computed property names*.

```
handleInputChange(event) {
  this.setState({ [event.target.id]: event.target.value })
}
```

[↑ Back to Top](#)

43. What would be the common mistake of function being called every time the component renders?

You need to make sure that function is not being called while passing the function as a parameter.

```
render() {
  // Wrong: handleClick is called instead of passed as a reference!
  return <button onClick={this.handleClick()}>'Click Me'</button>
}
```

Instead, pass the function itself without parenthesis:

```
render() {
  // Correct: handleClick is passed as a reference!
  return <button onClick={this.handleClick}>'Click Me'</button>
}
```

[↑ Back to Top](#)

44. Is lazy function supports named exports?

No, currently `React.lazy` function supports default exports only. If you would like to import modules which are named exports, you can create an intermediate module that reexports it as the default. It also ensures that tree shaking keeps working and don't pull

unused components. Let's take a component file which exports multiple named components,

```
// MoreComponents.js
export const SomeComponent = /* ... */;
export const UnusedComponent = /* ... */;
```

and reexport `MoreComponents.js` components in an intermediate file `IntermediateComponent.js`

```
// IntermediateComponent.js
export { SomeComponent as default } from "./MoreComponents.js";
```

Now you can import the module using `lazy` function as below,

```
import React, { lazy } from 'react';
const SomeComponent = lazy(() => import("./IntermediateComponent.js"));
```

[↑ Back to Top](#)

45. Why React uses `className` over `class` attribute?

`class` is a keyword in JavaScript, and JSX is an extension of JavaScript. That's the principal reason why React uses `className` instead of `class`. Pass a string as the `className` prop.

```
render() {
  return <span className={'menu navigation-menu'}>{'Menu'}</span>
}
```

[↑ Back to Top](#)

46. What are fragments?

It's common pattern in React which is used for a component to return multiple elements. *Fragments* let you group a list of children without adding extra nodes to the DOM.

```
render() {
  return (
    <React.Fragment>
      <ChildA />
      <ChildB />
      <ChildC />
    </React.Fragment>
  )
}
```

There is also a *shorter syntax*, but it's not supported in many tools:

```
render() {
  return (
    <>
      <ChildA />
      <ChildB />
      <ChildC />
    </>
  )
}
```

 [Back to Top](#)

47. Why fragments are better than container divs?

Below are the list of reasons,

- i. Fragments are a bit faster and use less memory by not creating an extra DOM node.
This only has a real benefit on very large and deep trees.
- ii. Some CSS mechanisms like *Flexbox* and *CSS Grid* have a special parent-child relationships, and adding divs in the middle makes it hard to keep the desired layout.
- iii. The DOM Inspector is less cluttered.

 [Back to Top](#)

48. What are portals in React?

Portal is a recommended way to render children into a DOM node that exists outside the DOM hierarchy of the parent component.

```
ReactDOM.createPortal(child, container)
```

The first argument is any render-able React child, such as an element, string, or fragment. The second argument is a DOM element.

 [Back to Top](#)

49. What are stateless components?

If the behaviour is independent of its state then it can be a stateless component. You can use either a function or a class for creating stateless components. But unless you need to use a lifecycle hook in your components, you should go for function components. There are a lot of benefits if you decide to use function components here; they are easy to write, understand, and test, a little faster, and you can avoid the `this` keyword altogether.

[↑ Back to Top](#)

50. What are stateful components?

If the behaviour of a component is dependent on the *state* of the component then it can be termed as stateful component. These *stateful components* are always *class components* and have a state that gets initialized in the `constructor`.

```
class App extends Component {
  constructor(props) {
    super(props)
    this.state = { count: 0 }
  }

  render() {
    // ...
  }
}
```

React 16.8 Update:

Hooks let you use state and other React features without writing classes.

The Equivalent Functional Component

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

const App = (props) => {
  const [count, setCount] = useState(0);

  return (
    // JSX
  )
}
```

[↑ Back to Top](#)

51. How to apply validation on props in React?

When the application is running in *development mode*, React will automatically check all props that we set on components to make sure they have *correct type*. If the type is incorrect, React will generate warning messages in the console. It's disabled in *production mode* due to performance impact. The mandatory props are defined with `isRequired`.

The set of predefined prop types:

- i. `PropTypes.number`
- ii. `PropTypes.string`
- iii. `PropTypes.array`
- iv. `PropTypes.object`
- v. `PropTypes.func`
- vi. `PropTypes.node`
- vii. `PropTypes.element`
- viii. `PropTypes.bool`
- ix. `PropTypes.symbol`
- X. `PropTypes.any`

We can define `propTypes` for `User` component as below:

```
import React from 'react'
import PropTypes from 'prop-types'

class User extends React.Component {
  static propTypes = {
    name: PropTypes.string.isRequired,
    age: PropTypes.number.isRequired
  }

  render() {
    return (
      <>
        <h1>`Welcome, ${this.props.name}`</h1>
        <h2>`Age, ${this.props.age}`</h2>
      </>
    )
  }
}
```

Note: In React v15.5 `PropTypes` were moved from `React.PropTypes` to `prop-types` library.

The Equivalent Functional Component

```
import React from 'react'
```

```
import PropTypes from 'prop-types'

function User() {
  return (
    <>
      <h1>{`Welcome, ${this.props.name}`}</h1>
      <h2>{`Age, ${this.props.age}`}</h2>
    </>
  )
}

User.propTypes = {
  name: PropTypes.string.isRequired,
  age: PropTypes.number.isRequired
}
```

 [Back to Top](#)

52. What are the advantages of React?

Below are the list of main advantages of React,

- i. Increases the application's performance with *Virtual DOM*.
- ii. JSX makes code easy to read and write.
- iii. It renders both on client and server side (*SSR*).
- iv. Easy to integrate with frameworks (Angular, Backbone) since it is only a view library.
- v. Easy to write unit and integration tests with tools such as Jest.

 [Back to Top](#)

53. What are the limitations of React?

Apart from the advantages, there are few limitations of React too,

- i. React is just a view library, not a full framework.
- ii. There is a learning curve for beginners who are new to web development.
- iii. Integrating React into a traditional MVC framework requires some additional configuration.
- iv. The code complexity increases with inline templating and JSX.
- v. Too many smaller components leading to over engineering or boilerplate.

 [Back to Top](#)

54. What are error boundaries in React v16?

Error boundaries are components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of the component tree that crashed.

A class component becomes an error boundary if it defines a new lifecycle method called `componentDidCatch(error, info)` or `static getDerivedStateFromError()` :

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props)
    this.state = { hasError: false }
  }

  componentDidCatch(error, info) {
    // You can also log the error to an error reporting service
    logErrorToMyService(error, info)
  }

  static getDerivedStateFromError(error) {
    // Update state so the next render will show the fallback UI.
    return { hasError: true };
  }

  render() {
    if (this.state.hasError) {
      // You can render any custom fallback UI
      return <h1>'Something went wrong.'</h1>
    }
    return this.props.children
  }
}
```

After that use it as a regular component:

```
<ErrorBoundary>
  <MyWidget />
</ErrorBoundary>
```

 [Back to Top](#)

55. How error boundaries handled in React v15?

React v15 provided very basic support for *error boundaries* using `unstable_handleError` method. It has been renamed to `componentDidCatch` in React v16.

 [Back to Top](#)

56. What are the recommended ways for static type checking?

Normally we use *PropTypes library* (`React.PropTypes` moved to a `prop-types` package since React v15.5) for *type checking* in the React applications. For large code bases, it is recommended to use *static type checkers* such as Flow or TypeScript, that perform type checking at compile time and provide auto-completion features.

 [Back to Top](#)

57. What is the use of `react-dom` package?

The `react-dom` package provides *DOM-specific methods* that can be used at the top level of your app. Most of the components are not required to use this module. Some of the methods of this package are:

- i. `render()`
- ii. `hydrate()`
- iii. `unmountComponentAtNode()`
- iv. `findDOMNode()`
- v. `createPortal()`

 [Back to Top](#)

58. What is the purpose of render method of `react-dom` ?

This method is used to render a React element into the DOM in the supplied container and return a reference to the component. If the React element was previously rendered into container, it will perform an update on it and only mutate the DOM as necessary to reflect the latest changes.

```
ReactDOM.render(element, container[, callback])
```

If the optional callback is provided, it will be executed after the component is rendered or updated.

 [Back to Top](#)

59. What is ReactDOMServer?

The `ReactDOMServer` object enables you to render components to static markup (typically used on node server). This object is mainly used for *server-side rendering* (SSR). The following methods can be used in both the server and browser environments:

- i. `renderToString()`
- ii. `renderToStaticMarkup()`

For example, you generally run a Node-based web server like Express, Hapi, or Koa, and you call `renderToString` to render your root component to a string, which you then send as response.

```
// using Express
import { renderToString } from 'react-dom/server'
import MyPage from './MyPage'

app.get('/', (req, res) => {
  res.write('<!DOCTYPE html><html><head><title>My Page</title></head><body>')
  res.write('<div id="content">')
  res.write(renderToString(<MyPage/>))
  res.write('</div></body></html>')
  res.end()
})
```

[↑ Back to Top](#)

60. How to use innerHTML in React?

The `dangerouslySetInnerHTML` attribute is React's replacement for using `innerHTML` in the browser DOM. Just like `innerHTML`, it is risky to use this attribute considering cross-site scripting (XSS) attacks. You just need to pass a `__html` object as key and HTML text as value.

In this example `MyComponent` uses `dangerouslySetInnerHTML` attribute for setting HTML markup:

```
function createMarkup() {
  return { __html: 'First &nbsp; Second' }
}

function MyComponent() {
  return <div dangerouslySetInnerHTML={createMarkup()} />
}
```

[↑ Back to Top](#)

61. How to use styles in React?

The `style` attribute accepts a JavaScript object with camelCased properties rather than a CSS string. This is consistent with the DOM style JavaScript property, is more efficient, and prevents XSS security holes.

```
const divStyle = {
  color: 'blue',
  backgroundImage: 'url(' + imgUrl + ')'
```

```
};

function HelloWorldComponent() {
  return <div style={divStyle}>Hello World!</div>
}
```

Style keys are camelCased in order to be consistent with accessing the properties on DOM nodes in JavaScript (e.g. `node.style.backgroundImage`).

[↑ Back to Top](#)

62. How events are different in React?

Handling events in React elements has some syntactic differences:

- i. React event handlers are named using camelCase, rather than lowercase.
- ii. With JSX you pass a function as the event handler, rather than a string.

[↑ Back to Top](#)

63. What will happen if you use `setState()` in constructor?

When you use `setState()` , then apart from assigning to the object state React also re-renders the component and all its children. You would get error like this: *Can only update a mounted or mounting component*. So we need to use `this.state` to initialize variables inside constructor.

[↑ Back to Top](#)

64. What is the impact of indexes as keys?

Keys should be stable, predictable, and unique so that React can keep track of elements.

In the below code snippet each element's key will be based on ordering, rather than tied to the data that is being represented. This limits the optimizations that React can do.

```
{todos.map((todo, index) =>
  <Todo
    {...todo}
    key={index}
  />
)}
```

If you use element data for unique key, assuming `todo.id` is unique to this list and stable, React would be able to reorder elements without needing to reevaluate them as much.

```
{todos.map((todo) =>
```

```
<Todo {...todo}
       key={todo.id} />
)}
```

[↑ Back to Top](#)

65. Is it good to use `setState()` in `componentWillMount()` method?

Yes, it is safe to use `setState()` inside `componentWillMount()` method. But at the same it is recommended to avoid async initialization in `componentWillMount()` lifecycle method. `componentWillMount()` is invoked immediately before mounting occurs. It is called before `render()`, therefore setting state in this method will not trigger a re-render. Avoid introducing any side-effects or subscriptions in this method. We need to make sure async calls for component initialization happened in `componentDidMount()` instead of `componentWillMount()`.

```
componentDidMount() {
  axios.get(`api/todos`)
    .then((result) => {
      this.setState({
        messages: [...result.data]
      })
    })
}
```

[↑ Back to Top](#)

66. What will happen if you use props in initial state?

If the props on the component are changed without the component being refreshed, the new prop value will never be displayed because the constructor function will never update the current state of the component. The initialization of state from props only runs when the component is first created.

The below component won't display the updated input value:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props)

    this.state = {
      records: [],
      inputValue: this.props.inputValue
    };
  }
}
```

```
    render() {
      return <div>{this.state inputValue}</div>
    }
}
```

Using props inside render method will update the value:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props)

    this.state = {
      record: []
    }
  }

  render() {
    return <div>{this.props.inputValue}</div>
  }
}
```

 [Back to Top](#)

67. How do you conditionally render components?

In some cases you want to render different components depending on some state. JSX does not render `false` or `undefined`, so you can use conditional *short-circuiting* to render a given part of your component only if a certain condition is true.

```
const MyComponent = ({ name, address }) => (
  <div>
    <h2>{name}</h2>
    {address &&
      <p>{address}</p>
    }
  </div>
)
```

If you need an `if-else` condition then use *ternary operator*.

```
const MyComponent = ({ name, address }) => (
  <div>
    <h2>{name}</h2>
    {address
      ? <p>{address}</p>
      : <p>{'Address is not available'}</p>
    }
  </div>
)
```

)

[↑ Back to Top](#)

68. Why we need to be careful when spreading props on DOM elements?

When we *spread props* we run into the risk of adding unknown HTML attributes, which is a bad practice. Instead we can use prop destructuring with `...rest` operator, so it will add only required props.

For example,

```
const ComponentA = () =>
  <ComponentB isDisplay={true} className={'componentStyle'} />

const ComponentB = ({ isDisplay, ...domProps }) =>
  <div {...domProps}>{'ComponentB'}</div>
```

[↑ Back to Top](#)

69. How you use decorators in React?

You can *decorate* your *class* components, which is the same as passing the component into a function. **Decorators** are flexible and readable way of modifying component functionality.

```
@setTitle('Profile')
class Profile extends React.Component {
  //....
}

/*
  title is a string that will be set as a document title
  WrappedComponent is what our decorator will receive when
  put directly above a component class as seen in the example above
*/
const setTitle = (title) => (WrappedComponent) => {
  return class extends React.Component {
    componentDidMount() {
      document.title = title
    }

    render() {
      return <WrappedComponent {...this.props} />
    }
  }
}
```

Note: Decorators are a feature that didn't make it into ES7, but are currently a *stage 2 proposal*.

 [Back to Top](#)

70. How do you memoize a component?

There are memoize libraries available which can be used on function components.

For example `moize` library can memoize the component in another component.

```
import moize from 'moize'
import Component from './components/Component' // this module exports a non-memoized component

const MemoizedFoo = moize.react(Component)

const Consumer = () => {
  <div>
    {'I will memoize the following entry:'}
    <MemoizedFoo/>
  </div>
}
```

Update: Since React v16.6.0, we have a `React.memo`. It provides a higher order component which memoizes component unless the props change. To use it, simply wrap the component using `React.memo` before you use it.

```
const MemoComponent = React.memo(function MemoComponent(props) {
  /* render using props */
});
OR
export default React.memo(MyFunctionComponent);
```

 [Back to Top](#)

71. How you implement Server Side Rendering or SSR?

React is already equipped to handle rendering on Node servers. A special version of the DOM renderer is available, which follows the same pattern as on the client side.

```
import ReactDOMServer from 'react-dom/server'
import App from './App'

ReactDOMServer.renderToString(<App />)
```

This method will output the regular HTML as a string, which can be then placed inside a page body as part of the server response. On the client side, React detects the pre-rendered content and seamlessly picks up where it left off.

[↑ Back to Top](#)

72. How to enable production mode in React?

You should use Webpack's `DefinePlugin` method to set `NODE_ENV` to `production`, by which it strips out things like propType validation and extra warnings. Apart from this, if you minify the code, for example, Uglify's dead-code elimination to strip out development only code and comments, it will drastically reduce the size of your bundle.

[↑ Back to Top](#)

73. What is CRA and its benefits?

The `create-react-app` CLI tool allows you to quickly create & run React applications with no configuration step.

Let's create Todo App using *CRA*:

```
# Installation
$ npm install -g create-react-app

# Create new project
$ create-react-app todo-app
$ cd todo-app

# Build, test and run
$ npm run build
$ npm run test
$ npm start
```

It includes everything we need to build a React app:

- i. React, JSX, ES6, and Flow syntax support.
- ii. Language extras beyond ES6 like the object spread operator.
- iii. Autoprefixed CSS, so you don't need `-webkit-` or other prefixes.
- iv. A fast interactive unit test runner with built-in support for coverage reporting.
- v. A live development server that warns about common mistakes.
- vi. A build script to bundle JS, CSS, and images for production, with hashes and sourcemaps.

[↑ Back to Top](#)

74. What is the lifecycle methods order in mounting?

The lifecycle methods are called in the following order when an instance of a component is being created and inserted into the DOM.

- i. constructor()
- ii. static getDerivedStateFromProps()
- iii. render()
- iv. componentDidMount()

 [Back to Top](#)

75. What are the lifecycle methods going to be deprecated in React v16?

The following lifecycle methods going to be unsafe coding practices and will be more problematic with async rendering.

- i. componentWillMount()
- ii. componentWillReceiveProps()
- iii. componentWillUpdate()

Starting with React v16.3 these methods are aliased with `UNSAFE_` prefix, and the unprefixed version will be removed in React v17.

 [Back to Top](#)

76. What is the purpose of `getDerivedStateFromProps()` lifecycle method?

The new static `getDerivedStateFromProps()` lifecycle method is invoked after a component is instantiated as well as before it is re-rendered. It can return an object to update state, or `null` to indicate that the new props do not require any state updates.

```
class MyComponent extends React.Component {  
  static getDerivedStateFromProps(props, state) {  
    // ...  
  }  
}
```

This lifecycle method along with `componentDidUpdate()` covers all the use cases of `componentWillReceiveProps()`.

 [Back to Top](#)

77. What is the purpose of `getSnapshotBeforeUpdate()` lifecycle method?

The new `getSnapshotBeforeUpdate()` lifecycle method is called right before DOM updates. The return value from this method will be passed as the third parameter to `componentDidUpdate()`.

```
class MyComponent extends React.Component {  
  getSnapshotBeforeUpdate(prevProps, prevState) {  
    // ...  
  }  
}
```

This lifecycle method along with `componentDidUpdate()` covers all the use cases of `componentWillUpdate()`.

 [Back to Top](#)

78. Do Hooks replace render props and higher order components?

Both render props and higher-order components render only a single child but in most of the cases Hooks are a simpler way to serve this by reducing nesting in your tree.

 [Back to Top](#)

79. What is the recommended way for naming components?

It is recommended to name the component by reference instead of using `displayName`.

Using `displayName` for naming component:

```
export default React.createClass({  
  displayName: 'TodoApp',  
  // ...  
})
```

The **recommended** approach:

```
export default class TodoApp extends React.Component {  
  // ...  
}
```

 [Back to Top](#)

80. What is the recommended ordering of methods in component class?

Recommended ordering of methods from *mounting* to *render stage*:

- i. static methods

- ii. constructor()
- iii. getChildContext()
- iv. componentWillMount()
- v. componentDidMount()
- vi. componentWillReceiveProps()
- vii. shouldComponentUpdate()
- viii. componentWillUpdate()
- ix. componentDidUpdate()
- x. componentWillUnmount()
- xi. click handlers or event handlers like onClickSubmit() or onChangeDescription()
- xii. getter methods for render like getSelectReason() or getFooterContent()
- xiii. optional render methods like renderNavigation() or renderProfilePicture()
- xiv. render()

 [Back to Top](#)

81. What is a switching component?

A *switching component* is a component that renders one of many components. We need to use object to map prop values to components.

For example, a switching component to display different pages based on page prop:

```

import HomePage from './HomePage'
import AboutPage from './AboutPage'
import ServicesPage from './ServicesPage'
import ContactPage from './ContactPage'

const PAGES = {
  home: HomePage,
  about: AboutPage,
  services: ServicesPage,
  contact: ContactPage
}

const Page = (props) => {
  const Handler = PAGES[props.page] || ContactPage

  return <Handler {...props} />
}

// The keys of the PAGES object can be used in the prop types to catch dev-time er
Page.propTypes = {
  page: PropTypes.oneOf(Object.keys(PAGES)).isRequired
}

```

82. Why we need to pass a function to `setState()`?

The reason behind for this is that `setState()` is an asynchronous operation. React batches state changes for performance reasons, so the state may not change immediately after `setState()` is called. That means you should not rely on the current state when calling `setState()` since you can't be sure what that state will be. The solution is to pass a function to `setState()`, with the previous state as an argument. By doing this you can avoid issues with the user getting the old state value on access due to the asynchronous nature of `setState()`.

Let's say the initial count value is zero. After three consecutive increment operations, the value is going to be incremented only by one.

```
// assuming this.state.count === 0
this.setState({ count: this.state.count + 1 })
this.setState({ count: this.state.count + 1 })
this.setState({ count: this.state.count + 1 })
// this.state.count === 1, not 3
```

If we pass a function to `setState()`, the count gets incremented correctly.

```
this.setState((prevState, props) => ({
  count: prevState.count + props.increment
}))
// this.state.count === 3 as expected
```

(OR)

Why function is preferred over object for `setState()` ?

React may batch multiple `setState()` calls into a single update for performance. Because `this.props` and `this.state` may be updated asynchronously, you should not rely on their values for calculating the next state.

This counter example will fail to update as expected:

```
// Wrong
this.setState({
  counter: this.state.counter + this.props.increment,
})
```

The preferred approach is to call `setState()` with function rather than object. That function will receive the previous state as the first argument, and the props at the time the update is applied as the second argument.

```
// Correct
this.setState((prevState, props) => {
  counter: prevState.counter + props.increment
})
```

 [Back to Top](#)

83. What is strict mode in React?

`React.StrictMode` is a useful component for highlighting potential problems in an application. Just like `<Fragment>`, `<StrictMode>` does not render any extra DOM elements. It activates additional checks and warnings for its descendants. These checks apply for *development mode* only.

```
import React from 'react'

function ExampleApplication() {
  return (
    <div>
      <Header />
      <React.StrictMode>
        <div>
          <ComponentOne />
          <ComponentTwo />
        </div>
      </React.StrictMode>
      <Footer />
    </div>
  )
}
```

In the example above, the *strict mode* checks apply to `<ComponentOne>` and `<ComponentTwo>` components only.

 [Back to Top](#)

84. What are React Mixins?

Mixins are a way to totally separate components to have a common functionality. Mixins **should not be used** and can be replaced with *higher-order components* or *decorators*.

One of the most commonly used mixins is `PureRenderMixin`. You might be using it in some components to prevent unnecessary re-renders when the props and state are shallowly equal to the previous props and state:

```
const PureRenderMixin = require('react-addons-pure-render-mixin')

const Button = React.createClass({
  mixins: [PureRenderMixin],
  // ...
})
```

 [Back to Top](#)

85. Why is `isMounted()` an anti-pattern and what is the proper solution?

The primary use case for `isMounted()` is to avoid calling `setState()` after a component has been unmounted, because it will emit a warning.

```
if (this.isMounted()) {
  this.setState({...})
}
```

Checking `isMounted()` before calling `setState()` does eliminate the warning, but it also defeats the purpose of the warning. Using `isMounted()` is a code smell because the only reason you would check is because you think you might be holding a reference after the component has unmounted.

An optimal solution would be to find places where `setState()` might be called after a component has unmounted, and fix them. Such situations most commonly occur due to callbacks, when a component is waiting for some data and gets unmounted before the data arrives. Ideally, any callbacks should be canceled in `componentWillUnmount()`, prior to unmounting.

 [Back to Top](#)

86. What are the Pointer Events supported in React?

Pointer Events provide a unified way of handling all input events. In the old days we had a mouse and respective event listeners to handle them but nowadays we have many devices which don't correlate to having a mouse, like phones with touch surface or pens. We need to remember that these events will only work in browsers that support the *Pointer Events* specification.

The following event types are now available in *React DOM*:

- i. onPointerDown
- ii. onPointerMove
- iii. onPointerUp
- iv. onPointerCancel
- v. onGotPointerCapture
- vi. onLostPointerCapture
- vii. onPointerEnter
- viii. onPointerLeave
- ix. onPointerOver
- X. onPointerOut

 [Back to Top](#)

87. Why should component names start with capital letter?

If you are rendering your component using JSX, the name of that component has to begin with a capital letter otherwise React will throw an error as unrecognized tag. This convention is because only HTML elements and SVG tags can begin with a lowercase letter.

```
class SomeComponent extends Component {
  // Code goes here
}
```

You can define component class which name starts with lowercase letter, but when it's imported it should have capital letter. Here lowercase is fine:

```
class myComponent extends Component {
  render() {
    return <div />
  }
}

export default myComponent
```

While when imported in another file it should start with capital letter:

```
import MyComponent from './MyComponent'
```

What are the exceptions on React component naming?

The component names should start with a uppercase letter but there are few exceptions on this convention. The lowercase tag names with a dot (property

accessors) are still considered as valid component names.

For example the below tag can be compiled to a valid component,

```
```js harmony
render(){
 return (
 <obj.component /> // `React.createElement(obj.component)`
)
}
```

```

 [Back to Top](#)

88. Are custom DOM attributes supported in React v16?

Yes. In the past, React used to ignore unknown DOM attributes. If you wrote JSX with an attribute that React doesn't recognize, React would just skip it.

For example, let's take a look at the below attribute:

```
<div mycustomattribute={'something'} />
```

Would render an empty div to the DOM with React v15:

```
<div />
```

In React v16 any unknown attributes will end up in the DOM:

```
<div mycustomattribute='something' />
```

This is useful for supplying browser-specific non-standard attributes, trying new DOM APIs, and integrating with opinionated third-party libraries.

 [Back to Top](#)

89. What is the difference between constructor and getInitialState?

You should initialize state in the constructor when using ES6 classes, and `getInitialState()` method when using `React.createClass()`.

Using ES6 classes:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props)
```

```
        this.state = { /* initial state */ }
    }
}
```

Using `React.createClass()`:

```
const MyComponent = React.createClass({
  getInitialState() {
    return { /* initial state */ }
  }
})
```

Note: `React.createClass()` is deprecated and removed in React v16. Use plain JavaScript classes instead.

[↑ Back to Top](#)

90. Can you force a component to re-render without calling `setState`?

By default, when your component's state or props change, your component will re-render. If your `render()` method depends on some other data, you can tell React that the component needs re-rendering by calling `forceUpdate()`.

```
component.forceUpdate(callback)
```

It is recommended to avoid all uses of `forceUpdate()` and only read from `this.props` and `this.state` in `render()`.

[↑ Back to Top](#)

91. What is the difference between `super()` and `super(props)` in React using ES6 classes?

When you want to access `this.props` in `constructor()` then you should pass props to `super()` method.

Using `super(props)`:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props)
    console.log(this.props) // { name: 'John', ... }
  }
}
```

Using `super()`:

```
class MyComponent extends React.Component {  
  constructor(props) {  
    super()  
    console.log(this.props) // undefined  
  }  
}
```

Outside `constructor()` both will display same value for `this.props`.

 [Back to Top](#)

92. How to loop inside JSX?

You can simply use `Array.prototype.map` with ES6 *arrow function* syntax.

For example, the `items` array of objects is mapped into an array of components:

```
<tbody>  
  {items.map(item => <SomeComponent key={item.id} name={item.name} />)}  
</tbody>
```

But you can't iterate using `for` loop:

```
<tbody>  
  for (let i = 0; i < items.length; i++) {  
    <SomeComponent key={items[i].id} name={items[i].name} />  
  }  
</tbody>
```

This is because JSX tags are transpiled into *function calls*, and you can't use statements inside expressions. This may change thanks to `do` expressions which are *stage 1 proposal*.

 [Back to Top](#)

93. How do you access props in attribute quotes?

React (or JSX) doesn't support variable interpolation inside an attribute value. The below representation won't work:

```
<img className='image' src='images/{this.props.image}' />
```

But you can put any JS expression inside curly braces as the entire attribute value. So the below expression works:

```
<img className='image' src={'images/' + this.props.image} />
```

Using *template strings* will also work:

```
<img className='image' src={`images/${this.props.image}`}>
```

 [Back to Top](#)

94. What is React propotype array with shape?

If you want to pass an array of objects to a component with a particular shape then use `React.PropTypes.shape()` as an argument to `React.PropTypes.arrayOf()`.

```
ReactComponent.propTypes = {
  arrayWithShape: React.PropTypes.arrayOf(React.PropTypes.shape({
    color: React.PropTypes.string.isRequired,
    fontSize: React.PropTypes.number.isRequired
  })).isRequired
}
```

 [Back to Top](#)

95. How to conditionally apply class attributes?

You shouldn't use curly braces inside quotes because it is going to be evaluated as a string.

```
<div className="btn-panel {this.props.visible ? 'show' : 'hidden'}">
```

Instead you need to move curly braces outside (don't forget to include spaces between class names):

```
<div className={'btn-panel ' + (this.props.visible ? 'show' : 'hidden')}>
```

Template strings will also work:

```
<div className={`btn-panel ${this.props.visible ? 'show' : 'hidden'}`}>
```

 [Back to Top](#)

96. What is the difference between React and ReactDOM?

The `react` package contains `React.createElement()` , `React.Component` , `React.Children` , and other helpers related to elements and component classes. You can think of these as the isomorphic or universal helpers that you need to build components. The `react-dom` package contains `ReactDOM.render()` , and in `react-dom/server` we have *server-side rendering* support with `ReactDOMServer.renderToString()` and `ReactDOMServer.renderToStaticMarkup()` .

 [Back to Top](#)

97. Why ReactDOM is separated from React?

The React team worked on extracting all DOM-related features into a separate library called *ReactDOM*. React v0.14 is the first release in which the libraries are split. By looking at some of the packages, `react-native` , `react-art` , `react-canvas` , and `react-three` , it has become clear that the beauty and essence of React has nothing to do with browsers or the DOM.

To build more environments that React can render to, React team planned to split the main React package into two: `react` and `react-dom` . This paves the way to writing components that can be shared between the web version of React and React Native.

 [Back to Top](#)

98. How to use React label element?

If you try to render a `<label>` element bound to a text input using the standard `for` attribute, then it produces HTML missing that attribute and prints a warning to the console.

```
<label for={'user'}>{'User'}</label>
<input type={'text'} id={'user'} />
```

Since `for` is a reserved keyword in JavaScript, use `htmlFor` instead.

```
<label htmlFor={'user'}>{'User'}</label>
<input type={'text'} id={'user'} />
```

 [Back to Top](#)

99. How to combine multiple inline style objects?

You can use *spread operator* in regular React:

```
<button style={{...styles.panel.button, ...styles.panel.submitButton}}>{'Submit'}
```

If you're using React Native then you can use the array notation:

```
<button style={[styles.panel.button, styles.panel.submitButton]}>'Submit'</button>
```

 [Back to Top](#)

100. How to re-render the view when the browser is resized?

You can listen to the `resize` event in `componentDidMount()` and then update the dimensions (`width` and `height`). You should remove the listener in `componentWillUnmount()` method.

```
class WindowDimensions extends React.Component {
  constructor(props){
    super(props);
    this.updateDimensions = this.updateDimensions.bind(this);
  }

  componentWillMount() {
    this.updateDimensions()
  }

  componentDidMount() {
    window.addEventListener('resize', this.updateDimensions)
  }

  componentWillUnmount() {
    window.removeEventListener('resize', this.updateDimensions)
  }

  updateDimensions() {
    this.setState({width: window.innerWidth, height: window.innerHeight})
  }

  render() {
    return <span>{this.state.width} x {this.state.height}</span>
  }
}
```

 [Back to Top](#)

101. What is the difference between `setState()` and `replaceState()` methods?

When you use `setState()` the current and previous states are merged. `replaceState()` throws out the current state, and replaces it with only what you provide. Usually `setState()` is used unless you really need to remove all previous keys for some reason. You can also set state to `false / null` in `setState()` instead of using `replaceState()`.

[↑ Back to Top](#)

102. How to listen to state changes?

The `componentDidUpdate` lifecycle method will be called when state changes. You can compare provided state and props values with current state and props to determine if something meaningful changed.

```
componentDidUpdate(object nextProps, object prevState)
```

Note: The previous releases of ReactJS also uses `componentWillUpdate(object nextProps, object nextState)` for state changes. It has been deprecated in latest releases.

[↑ Back to Top](#)

103. What is the recommended approach of removing an array element in React state?

The better approach is to use `Array.prototype.filter()` method.

For example, let's create a `removeItem()` method for updating the state.

```
removeItem(index) {
  this.setState({
    data: this.state.data.filter((item, i) => i !== index)
  })
}
```

[↑ Back to Top](#)

104. Is it possible to use React without rendering HTML?

It is possible with latest version ($>=16.2$). Below are the possible options:

```
render() {
  return false
}
```

```
render() {
  return null
}
```

```
render() {
```

```
    return []
}

render() {
  return <React.Fragment></React.Fragment>
}

render() {
  return <></>
}
```

Returning `undefined` won't work.

[↑ Back to Top](#)

105. How to pretty print JSON with React?

We can use `<pre>` tag so that the formatting of the `JSON.stringify()` is retained:

```
const data = { name: 'John', age: 42 }

class User extends React.Component {
  render() {
    return (
      <pre>
        {JSON.stringify(data, null, 2)}
      </pre>
    )
  }
}

React.render(<User />, document.getElementById('container'))
```

[↑ Back to Top](#)

106. Why you can't update props in React?

The React philosophy is that props should be *immutable* and *top-down*. This means that a parent can send any prop values to a child, but the child can't modify received props.

[↑ Back to Top](#)

107. How to focus an input element on page load?

You can do it by creating `ref` for `input` element and using it in `componentDidMount()`:

```

class App extends React.Component{
  componentDidMount() {
    this.nameInput.focus()
  }

  render() {
    return (
      <div>
        <input
          defaultValue={'Won\'t focus'}
        />
        <input
          ref={(input) => this.nameInput = input}
          defaultValue={'Will focus'}
        />
      </div>
    )
  }
}

ReactDOM.render(<App />, document.getElementById('app'))

```

 [Back to Top](#)

108. What are the possible ways of updating objects in state?

i. Calling `setState()` with an object to merge with state:

- Using `Object.assign()` to create a copy of the object:

```

const user = Object.assign({}, this.state.user, { age: 42 })
this.setState({ user })

```

- Using *spread operator*:

```

const user = { ...this.state.user, age: 42 }
this.setState({ user })

```

ii. Calling `setState()` with a function:

```

this.setState(prevState => ({
  user: {
    ...prevState.user,
    age: 42
  }
}))

```

[↑ Back to Top](#)

110. How can we find the version of React at runtime in the browser?

You can use `React.version` to get the version.

```
const REACT_VERSION = React.version

ReactDOM.render(
  <div>`React version: ${REACT_VERSION}`</div>,
  document.getElementById('app')
)
```

[↑ Back to Top](#)

111. What are the approaches to include polyfills in your `create-react-app` ?

There are approaches to include polyfills in `create-react-app`,

i. Manual import from `core-js` :

Create a file called (something like) `polyfills.js` and import it into root `index.js` file. Run `npm install core-js` or `yarn add core-js` and import your specific required features.

```
import 'core-js/fn/array/find'
import 'core-js/fn/array/includes'
import 'core-js/fn/number/is-nan'
```

ii. Using Polyfill service:

Use the `polyfill.io` CDN to retrieve custom, browser-specific polyfills by adding this line to `index.html` :

```
<script src='https://cdn.polyfill.io/v2/polyfill.min.js?features=default,Arra
```

In the above script we had to explicitly request the `Array.prototype.includes` feature as it is not included in the default feature set.

[↑ Back to Top](#)

112. How to use https instead of http in `create-react-app`?

You just need to use `HTTPS=true` configuration. You can edit your `package.json` scripts section:

```
"scripts": {  
  "start": "set HTTPS=true && react-scripts start"  
}
```

or just run `set HTTPS=true && npm start`

[↑ Back to Top](#)

113. How to avoid using relative path imports in create-react-app?

Create a file called `.env` in the project root and write the import path:

```
NODE_PATH=src/app
```

After that restart the development server. Now you should be able to import anything inside `src/app` without relative paths.

[↑ Back to Top](#)

114. How to add Google Analytics for React Router?

Add a listener on the `history` object to record each page view:

```
history.listen(function (location) {  
  window.ga('set', 'page', location.pathname + location.search)  
  window.ga('send', 'pageview', location.pathname + location.search)  
})
```

[↑ Back to Top](#)

115. How to update a component every second?

You need to use `setInterval()` to trigger the change, but you also need to clear the timer when the component unmounts to prevent errors and memory leaks.

```
componentDidMount() {  
  this.interval = setInterval(() => this.setState({ time: Date.now() }), 1000)  
}  
  
componentWillUnmount() {  
  clearInterval(this.interval)  
}
```

[↑ Back to Top](#)

116. How do you apply vendor prefixes to inline styles in React?

React *does not* apply *vendor prefixes* automatically. You need to add vendor prefixes manually.

```
<div style={{
  transform: 'rotate(90deg)',
  WebkitTransform: 'rotate(90deg)', // note the capital 'W' here
  msTransform: 'rotate(90deg)' // 'ms' is the only lowercase vendor prefix
}} />
```

[↑ Back to Top](#)

117. How to import and export components using React and ES6?

You should use default for exporting the components

```
import React from 'react'
import User from 'user'

export default class MyProfile extends React.Component {
  render(){
    return (
      <User type="customer">
        //...
      </User>
    )
  }
}
```

With the export specifier, the MyProfile is going to be the member and exported to this module and the same can be imported without mentioning the name in other components.

[↑ Back to Top](#)

119. Why is a component constructor called only once?

React's *reconciliation* algorithm assumes that without any information to the contrary, if a custom component appears in the same place on subsequent renders, it's the same component as before, so reuses the previous instance rather than creating a new one.

[↑ Back to Top](#)

120. How to define constants in React?

You can use ES7 `static` field to define constant.

```
class MyComponent extends React.Component {  
  static DEFAULT_PAGINATION = 10  
}
```

Static fields are part of the *Class Fields* stage 3 proposal.

[↑ Back to Top](#)

121. How to programmatically trigger click event in React?

You could use the `ref` prop to acquire a reference to the underlying `HTMLInputElement` object through a callback, store the reference as a class property, then use that reference to later trigger a click from your event handlers using the `HTMLElement.click` method.

This can be done in two steps:

- i. Create ref in render method:

```
<input ref={input => this.inputElement = input} />
```

- ii. Apply click event in your event handler:

```
this.inputElement.click()
```

[↑ Back to Top](#)

122. Is it possible to use `async/await` in plain React?

If you want to use `async / await` in React, you will need *Babel* and [transform-async-to-generator](#) plugin. React Native ships with Babel and a set of transforms.

[↑ Back to Top](#)

123. What are the common folder structures for React?

There are two common practices for React project file structure.

- i. Grouping by features or routes:

One common way to structure projects is locate CSS, JS, and tests together, grouped by feature or route.

```
common/  
|   └── Avatar.js  
|   └── Avatar.css  
|   └── APIUtils.js
```

```
└ APIUtils.test.js
feed/
├ index.js
├ Feed.js
├ Feed.css
├ FeedStory.js
├ FeedStory.test.js
└ FeedAPI.js
profile/
├ index.js
├ Profile.js
├ ProfileHeader.js
├ ProfileHeader.css
└ ProfileAPI.js
```

ii. Grouping by file type:

Another popular way to structure projects is to group similar files together.

```
api/
├ APIUtils.js
├ APIUtils.test.js
├ ProfileAPI.js
└ UserAPI.js
components/
├ Avatar.js
├ Avatar.css
├ Feed.js
├ Feed.css
├ FeedStory.js
├ FeedStory.test.js
├ Profile.js
├ ProfileHeader.js
└ ProfileHeader.css
```

[↑ Back to Top](#)

124. What are the popular packages for animation?

React Transition Group and *React Motion* are popular animation packages in React ecosystem.

[↑ Back to Top](#)

125. What is the benefit of styles modules?

It is recommended to avoid hard coding style values in components. Any values that are likely to be used across different UI components should be extracted into their own modules.

For example, these styles could be extracted into a separate component:

```
export const colors = {
  white,
  black,
  blue
}

export const space = [
  0,
  8,
  16,
  32,
  64
]
```

And then imported individually in other components:

```
import { space, colors } from './styles'
```

 [Back to Top](#)

126. What are the popular React-specific linters?

ESLint is a popular JavaScript linter. There are plugins available that analyse specific code styles. One of the most common for React is an npm package called `eslint-plugin-react`. By default, it will check a number of best practices, with rules checking things from keys in iterators to a complete set of prop types.

Another popular plugin is `eslint-plugin-jsx-a11y`, which will help fix common issues with accessibility. As JSX offers slightly different syntax to regular HTML, issues with `alt` text and `tabindex`, for example, will not be picked up by regular plugins.

 [Back to Top](#)

127. How to make AJAX call and in which component lifecycle methods should I make an AJAX call?

You can use AJAX libraries such as Axios, jQuery AJAX, and the browser built-in `fetch`. You should fetch data in the `componentDidMount()` lifecycle method. This is so you can use `setState()` to update your component when the data is retrieved.

For example, the employees list fetched from API and set local state:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props)
```

```

    this.state = {
      employees: [],
      error: null
    }
  }

componentDidMount() {
  fetch('https://api.example.com/items')
    .then(res => res.json())
    .then(
      (result) => {
        this.setState({
          employees: result.items
        })
      },
      (error) => {
        this.setState({ error })
      }
    )
}

render() {
  const { error, employees } = this.state
  if (error) {
    return <div>Error: {error.message}</div>;
  } else {
    return (
      <ul>
        {employees.map(employee => (
          <li key={employee.name}>
            {employee.name}-{employee.experience}
          </li>
        ))}
      </ul>
    )
  }
}

```

 [Back to Top](#)

128. What are render props?

Render Props is a simple technique for sharing code between components using a prop whose value is a function. The below component uses render prop which returns a React element.

```

<DataProvider render={data => (
  <h1>`Hello ${data.target}`</h1>
)}>

```

Libraries such as React Router and DownShift are using this pattern.

React Router

 [Back to Top](#)

129. What is React Router?

React Router is a powerful routing library built on top of React that helps you add new screens and flows to your application incredibly quickly, all while keeping the URL in sync with what's being displayed on the page.

 [Back to Top](#)

130. How React Router is different from history library?

React Router is a wrapper around the `history` library which handles interaction with the browser's `window.history` with its browser and hash histories. It also provides memory history which is useful for environments that don't have global history, such as mobile app development (React Native) and unit testing with Node.

 [Back to Top](#)

131. What are the `<Router>` components of React Router v4?

React Router v4 provides below 3 `<Router>` components:

- i. `<BrowserRouter>`
- ii. `<HashRouter>`
- iii. `<MemoryRouter>`

The above components will create *browser*, *hash*, and *memory* history instances. React Router v4 makes the properties and methods of the `history` instance associated with your router available through the context in the `router` object.

 [Back to Top](#)

132. What is the purpose of `push()` and `replace()` methods of `history` ?

A history instance has two methods for navigation purpose.

- i. `push()`
- ii. `replace()`

If you think of the history as an array of visited locations, `push()` will add a new location to the array and `replace()` will replace the current location in the array with the new one.

 [Back to Top](#)

133. How do you programmatically navigate using React Router v4?

There are three different ways to achieve programmatic routing/navigation within components.

i. Using the `withRouter()` higher-order function:

The `withRouter()` higher-order function will inject the history object as a prop of the component. This object provides `push()` and `replace()` methods to avoid the usage of context.

```
import { withRouter } from 'react-router-dom' // this also works with 'react-router'

const Button = withRouter(({ history }) => (
  <button
    type='button'
    onClick={() => { history.push('/new-location') }}
  >
    {'Click Me!'}
  </button>
))


```

ii. Using `<Route>` component and render props pattern:

The `<Route>` component passes the same props as `withRouter()`, so you will be able to access the history methods through the history prop.

```
import { Route } from 'react-router-dom'

const Button = () => (
  <Route render={({ history }) => (
    <button
      type='button'
      onClick={() => { history.push('/new-location') }}
    >
      {'Click Me!'}
    </button>
  )} />
)


```

iii. Using context:

This option is not recommended and treated as unstable API.

```
const Button = (props, context) => (
  <button
    type='button'
    onClick={() => {
      context.history.push('/new-location')
    }}
  >
    {'Click Me!'}
  </button>
)

Button.contextTypes = {
  history: React.PropTypes.shape({
    push: React.PropTypes.func.isRequired
  })
}
```

 [Back to Top](#)

134. How to get query parameters in React Router v4?

The ability to parse query strings was taken out of React Router v4 because there have been user requests over the years to support different implementation. So the decision has been given to users to choose the implementation they like. The recommended approach is to use query strings library.

```
const queryString = require('query-string');
const parsed = queryString.parse(props.location.search);
```

You can also use `URLSearchParams` if you want something native:

```
const params = new URLSearchParams(props.location.search)
const foo = params.get('name')
```

You should use a *polyfill* for IE11.

 [Back to Top](#)

135. Why you get "Router may have only one child element" warning?

You have to wrap your Route's in a `<Switch>` block because `<Switch>` is unique in that it renders a route exclusively.

At first you need to add `Switch` to your imports:

```
import { Switch, Router, Route } from 'react-router'
```

Then define the routes within `<Switch>` block:

```
<Router>
  <Switch>
    <Route /* ... */ />
    <Route /* ... */ />
  </Switch>
</Router>
```

 [Back to Top](#)

136. How to pass params to `history.push` method in React Router v4?

While navigating you can pass props to the `history` object:

```
this.props.history.push({
  pathname: '/template',
  search: '?name=sudheer',
  state: { detail: response.data }
})
```

The `search` property is used to pass query params in `push()` method.

 [Back to Top](#)

137. How to implement **default** or **NotFound** page?

A `<Switch>` renders the first child `<Route>` that matches. A `<Route>` with no path always matches. So you just need to simply drop path attribute as below

```
<Switch>
  <Route exact path="/" component={Home}/>
  <Route path="/user" component={User}/>
  <Route component={NotFound} />
</Switch>
```

 [Back to Top](#)

138. How to get history on React Router v4?

Below are the list of steps to get history object on React Router v4,

i.

Create a module that exports a `history` object and import this module across the project.

For example, create `history.js` file:

```
import { createBrowserHistory } from 'history'

export default createBrowserHistory({
  /* pass a configuration object here if needed */
})
```

ii. You should use the `<Router>` component instead of built-in routers. Imported the above `history.js` inside `index.js` file:

```
import { Router } from 'react-router-dom'
import history from './history'
import App from './App'

ReactDOM.render(
  <Router history={history}>
    <App />
  </Router>
), holder)
```

iii. You can also use push method of `history` object similar to built-in history object:

```
// some-other-file.js
import history from './history'

history.push('/go-here')
```

 [Back to Top](#)

139. How to perform automatic redirect after login?

The `react-router` package provides `<Redirect>` component in React Router. Rendering a `<Redirect>` will navigate to a new location. Like server-side redirects, the new location will override the current location in the history stack.

```
import React, { Component } from 'react'
import { Redirect } from 'react-router'

export default class LoginComponent extends Component {
  render() {
    if (this.state.isLoggedIn === true) {
      return <Redirect to="/your/redirect/page" />
    } else {
```

```
        return <div>'Login Please'</div>
    }
}
}
```

React Internationalization

[↑ Back to Top](#)

140. What is React Intl?

The *React Intl* library makes internationalization in React straightforward, with off-the-shelf components and an API that can handle everything from formatting strings, dates, and numbers, to pluralization. React Intl is part of *FormatJS* which provides bindings to React via its components and API.

[↑ Back to Top](#)

141. What are the main features of React Intl?

Below are the main features of React Intl,

- i. Display numbers with separators.
- ii. Display dates and times correctly.
- iii. Display dates relative to "now".
- iv. Pluralize labels in strings.
- v. Support for 150+ languages.
- vi. Runs in the browser and Node.
- vii. Built on standards.

[↑ Back to Top](#)

142. What are the two ways of formatting in React Intl?

The library provides two ways to format strings, numbers, and dates:

- i. Using react components:

```
<FormattedMessage
  id={'account'}
  defaultMessage={'The amount is less than minimum balance.'}
/>
```

- 2. **Using an API:**

```

```javascript
const messages = defineMessages({
 accountMessage: {
 id: 'account',
 defaultMessage: 'The amount is less than minimum balance.',
 }
})

formatMessage(messages.accountMessage)
```

```

[↑ Back to Top](#)

143. How to use <FormattedMessage> as placeholder using React Intl?

The <FormattedMessage... /> components from `react-intl` return elements, not plain text, so they can't be used for placeholders, alt text, etc. In that case, you should use lower level API `formatMessage()`. You can inject the `intl` object into your component using `injectIntl()` higher-order component and then format the message using `formatMessage()` available on that object.

```

import React from 'react'
import { injectIntl, intlShape } from 'react-intl'

const MyComponent = ({ intl }) => {
  const placeholder = intl.formatMessage({id: 'messageId'})
  return <input placeholder={placeholder} />
}

MyComponent.propTypes = {
  intl: intlShape.isRequired
}

export default injectIntl(MyComponent)

```

[↑ Back to Top](#)

144. How to access current locale with React Intl?

You can get the current locale in any component of your application using `injectIntl()`:

```

import { injectIntl, intlShape } from 'react-intl'

const MyComponent = ({ intl }) => (
  <div>`The current locale is ${intl.locale}`</div>
)

```

```
MyComponent.propTypes = {
  intl: intlShape.isRequired
}

export default injectIntl(MyComponent)
```

[↑ Back to Top](#)

145. How to format date using React Intl?

The `injectIntl()` higher-order component will give you access to the `formatDate()` method via the props in your component. The method is used internally by instances of `FormattedDate` and it returns the string representation of the formatted date.

```
import { injectIntl, intlShape } from 'react-intl'

const stringDate = this.props.intl.formatDate(date, {
  year: 'numeric',
  month: 'numeric',
  day: 'numeric'
})

const MyComponent = ({intl}) => (
  <div>`The formatted date is ${stringDate}`</div>
)

MyComponent.propTypes = {
  intl: intlShape.isRequired
}

export default injectIntl(MyComponent)
```

React Testing

[↑ Back to Top](#)

146. What is Shallow Renderer in React testing?

Shallow rendering is useful for writing unit test cases in React. It lets you render a component *one level deep* and assert facts about what its render method returns, without worrying about the behavior of child components, which are not instantiated or rendered.

For example, if you have the following component:

```
function MyComponent() {
  return (
```

```

        <div>
            <span className={'heading'}>{'Title'}</span>
            <span className={'description'}>{'Description'}</span>
        </div>
    )
}

```

Then you can assert as follows:

```

import ShallowRenderer from 'react-test-renderer/shallow'

// in your test
const renderer = new ShallowRenderer()
renderer.render(<MyComponent />

const result = renderer.getRenderOutput()

expect(result.type).toBe('div')
expect(result.props.children).toEqual([
    <span className={'heading'}>{'Title'}</span>,
    <span className={'description'}>{'Description'}</span>
])

```

 [Back to Top](#)

147. What is TestRenderer package in React?

This package provides a renderer that can be used to render components to pure JavaScript objects, without depending on the DOM or a native mobile environment. This package makes it easy to grab a snapshot of the platform view hierarchy (similar to a DOM tree) rendered by a ReactDOM or React Native without using a browser or jsdom .

```

import TestRenderer from 'react-test-renderer'

const Link = ({page, children}) => <a href={page}>{children}</a>

const testRenderer = TestRenderer.create(
    <Link page='https://www.facebook.com/'>{'Facebook'}</Link>
)

console.log(testRenderer.toJSON())
// {
//   type: 'a',
//   props: { href: 'https://www.facebook.com/' },
//   children: [ 'Facebook' ]
// }

```

 [Back to Top](#)

148. What is the purpose of ReactTestUtils package?

ReactTestUtils are provided in the `with-addons` package and allow you to perform actions against a simulated DOM for the purpose of unit testing.

[↑ Back to Top](#)

149. What is Jest?

Jest is a JavaScript unit testing framework created by Facebook based on Jasmine and provides automated mock creation and a `jsdom` environment. It's often used for testing components.

[↑ Back to Top](#)

150. What are the advantages of Jest over Jasmine?

There are couple of advantages compared to Jasmine:

- Automatically finds tests to execute in your source code.
- Automatically mocks dependencies when running your tests.
- Allows you to test asynchronous code synchronously.
- Runs your tests with a fake DOM implementation (via `jsdom`) so that your tests can be run on the command line.
- Runs tests in parallel processes so that they finish sooner.

[↑ Back to Top](#)

151. Give a simple example of Jest test case

Let's write a test for a function that adds two numbers in `sum.js` file:

```
const sum = (a, b) => a + b

export default sum
```

Create a file named `sum.test.js` which contains actual test:

```
import sum from './sum'

test('adds 1 + 2 to equal 3', () => {
  expect(sum(1, 2)).toBe(3)
})
```

And then add the following section to your `package.json`:

```
{
  "scripts": {
    "test": "jest"
  }
}
```

Finally, run `yarn test` or `npm test` and Jest will print a result:

```
$ yarn test
PASS ./sum.test.js
✓ adds 1 + 2 to equal 3 (2ms)
```

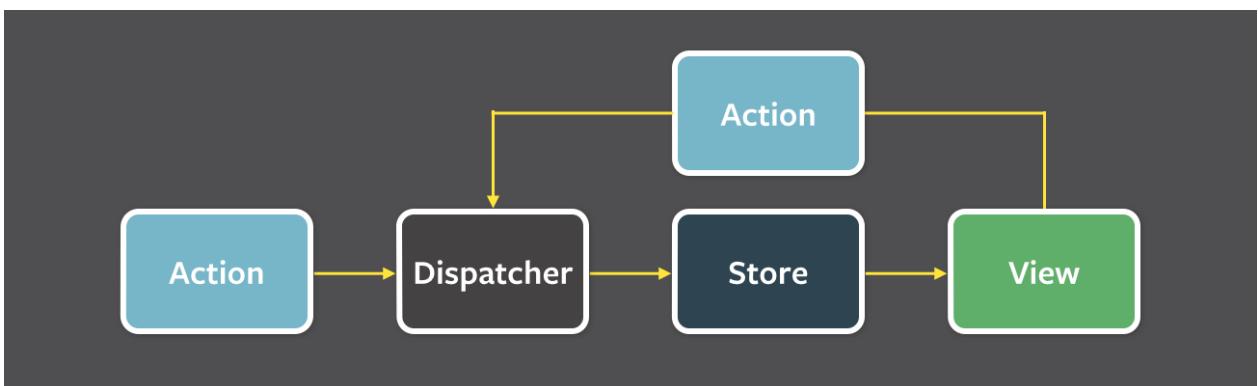
React Redux

[↑ Back to Top](#)

152. What is flux?

Flux is an *application design paradigm* used as a replacement for the more traditional MVC pattern. It is not a framework or a library but a new kind of architecture that complements React and the concept of Unidirectional Data Flow. Facebook uses this pattern internally when working with React.

The workflow between dispatcher, stores and views components with distinct inputs and outputs as follows:



[↑ Back to Top](#)

153. What is Redux?

Redux is a predictable state container for JavaScript apps based on the *Flux design pattern*. Redux can be used together with React, or with any other view library. It is tiny (about 2kB) and has no dependencies.

[↑ Back to Top](#)

154. What are the core principles of Redux?

Redux follows three fundamental principles:

- i. **Single source of truth:** The state of your whole application is stored in an object tree within a single store. The single state tree makes it easier to keep track of changes over time and debug or inspect the application.
- ii. **State is read-only:** The only way to change the state is to emit an action, an object describing what happened. This ensures that neither the views nor the network callbacks will ever write directly to the state.
- iii. **Changes are made with pure functions:** To specify how the state tree is transformed by actions, you write reducers. Reducers are just pure functions that take the previous state and an action as parameters, and return the next state.

 [Back to Top](#)

155. What are the downsides of Redux compared to Flux?

Instead of saying downsides we can say that there are few compromises of using Redux over Flux. Those are as follows:

- i. **You will need to learn to avoid mutations:** Flux is un-opinionated about mutating data, but Redux doesn't like mutations and many packages complementary to Redux assume you never mutate the state. You can enforce this with dev-only packages like `redux-immutable-state-invariant`, Immutable.js, or instructing your team to write non-mutating code.
- ii. **You're going to have to carefully pick your packages:** While Flux explicitly doesn't try to solve problems such as undo/redo, persistence, or forms, Redux has extension points such as middleware and store enhancers, and it has spawned a rich ecosystem.
- iii. **There is no nice Flow integration yet:** Flux currently lets you do very impressive static type checks which Redux doesn't support yet.

 [Back to Top](#)

156. What is the difference between `mapStateToProps()` and `mapDispatchToProps()` ?

`mapStateToProps()` is a utility which helps your component get updated state (which is updated by some other components):

```
const mapStateToProps = (state) => {
  return {
    todos: getVisibleTodos(state.todos, state.visibilityFilter)
  }
}
```

`mapDispatchToProps()` is a utility which will help your component to fire an action event (dispatching action which may cause change of application state):

```
const mapDispatchToProps = (dispatch) => {
  return {
    onTodoClick: (id) => {
      dispatch(toggleTodo(id))
    }
  }
}
```

Recommend always using the “object shorthand” form for the `mapDispatchToProps`

Redux wrap it in another function that looks like `(...args) => dispatch(onTodoClick(... args))`, and pass that wrapper function as a prop to your component.

```
const mapDispatchToProps = ({
  onTodoClick
})
```

 [Back to Top](#)

157. Can I dispatch an action in reducer?

Dispatching an action within a reducer is an **anti-pattern**. Your reducer should be *without side effects*, simply digesting the action payload and returning a new state object. Adding listeners and dispatching actions within the reducer can lead to chained actions and other side effects.

 [Back to Top](#)

158. How to access Redux store outside a component?

You just need to export the store from the module where it created with `createStore()`. Also, it shouldn't pollute the global window object.

```
store = createStore(myReducer)

export default store
```

[↑ Back to Top](#)

159. What are the drawbacks of MVW pattern?

- i. DOM manipulation is very expensive which causes applications to behave slow and inefficient.
- ii. Due to circular dependencies, a complicated model was created around models and views.
- iii. Lot of data changes happens for collaborative applications(like Google Docs).
- iv. No way to do undo (travel back in time) easily without adding so much extra code.

[↑ Back to Top](#)

160. Are there any similarities between Redux and RxJS?

These libraries are very different for very different purposes, but there are some vague similarities.

Redux is a tool for managing state throughout the application. It is usually used as an architecture for UIs. Think of it as an alternative to (half of) Angular. RxJS is a reactive programming library. It is usually used as a tool to accomplish asynchronous tasks in JavaScript. Think of it as an alternative to Promises. Redux uses the Reactive paradigm because the Store is reactive. The Store observes actions from a distance, and changes itself. RxJS also uses the Reactive paradigm, but instead of being an architecture, it gives you basic building blocks, Observables, to accomplish this pattern.

[↑ Back to Top](#)

161. How to dispatch an action on load?

You can dispatch an action in `componentDidMount()` method and in `render()` method you can verify the data.

```
class App extends Component {  
  componentDidMount() {  
    this.props.fetchData()  
  }  
  
  render() {  
    return this.props.isLoaded  
      ? <div>'Loaded'</div>  
      : <div>'Not Loaded'</div>  
  }  
}  
  
const mapStateToProps = (state) => ({  
  isLoaded: state.isLoaded  
})
```

```
const mapDispatchToProps = { fetchData }

export default connect(mapStateToProps, mapDispatchToProps)(App)
```

[↑ Back to Top](#)

162. How to use `connect()` from React Redux?

You need to follow two steps to use your store in your container:

- i. Use `mapStateToProps()` : It maps the state variables from your store to the props that you specify.
- ii. Connect the above props to your container: The object returned by the `mapStateToProps` function is connected to the container. You can import `connect()` from `react-redux`.

```
import React from 'react'
import { connect } from 'react-redux'

class App extends React.Component {
  render() {
    return <div>{this.props.containerData}</div>
  }
}

function mapStateToProps(state) {
  return { containerData: state.data }
}

export default connect(mapStateToProps)(App)
```

[↑ Back to Top](#)

163. How to reset state in Redux?

You need to write a *root reducer* in your application which delegate handling the action to the reducer generated by `combineReducers()`.

For example, let us take `rootReducer()` to return the initial state after `USER_LOGOUT` action. As we know, reducers are supposed to return the initial state when they are called with `undefined` as the first argument, no matter the action.

```
const appReducer = combineReducers({
  /* your app's top-level reducers */
})
```

```

const rootReducer = (state, action) => {
  if (action.type === 'USER_LOGOUT') {
    state = undefined
  }

  return appReducer(state, action)
}

```

In case of using `redux-persist`, you may also need to clean your storage. `redux-persist` keeps a copy of your state in a storage engine. First, you need to import the appropriate storage engine and then, to parse the state before setting it to undefined and clean each storage state key.

```

const appReducer = combineReducers({
  /* your app's top-level reducers */
})

const rootReducer = (state, action) => {
  if (action.type === 'USER_LOGOUT') {
    Object.keys(state).forEach(key => {
      storage.removeItem(`persist:${key}`)
    })
  }

  state = undefined
}

return appReducer(state, action)
}

```

 [Back to Top](#)

164. What's the purpose of `@` symbol in the Redux connect decorator?

The `@` symbol is in fact a JavaScript expression used to signify decorators. *Decorators* make it possible to annotate and modify classes and properties at design time.

Let's take an example setting up Redux without and with a decorator.

- Without decorator:

```

import React from 'react'
import * as actionCreators from './actionCreators'
import { bindActionCreators } from 'redux'
import { connect } from 'react-redux'

function mapStateToProps(state) {
  return { todos: state.todos }
}

```

```

function mapDispatchToProps(dispatch) {
  return { actions: bindActionCreators(actionCreators, dispatch) }
}

class MyApp extends React.Component {
  // ...define your main app here
}

export default connect(mapStateToProps, mapDispatchToProps)(MyApp)

```

- With decorator:

```

import React from 'react'
import * as actionCreators from './actionCreators'
import { bindActionCreators } from 'redux'
import { connect } from 'react-redux'

function mapStateToProps(state) {
  return { todos: state.todos }
}

function mapDispatchToProps(dispatch) {
  return { actions: bindActionCreators(actionCreators, dispatch) }
}

@connect(mapStateToProps, mapDispatchToProps)
export default class MyApp extends React.Component {
  // ...define your main app here
}

```

The above examples are almost similar except the usage of decorator. The decorator syntax isn't built into any JavaScript runtimes yet, and is still experimental and subject to change. You can use babel for the decorators support.

[↑ Back to Top](#)

165. What is the difference between React context and React Redux?

You can use **Context** in your application directly and is going to be great for passing down data to deeply nested components which what it was designed for.

Whereas **Redux** is much more powerful and provides a large number of features that the Context API doesn't provide. Also, React Redux uses context internally but it doesn't expose this fact in the public API.

[↑ Back to Top](#)

166. Why are Redux state functions called reducers?

Reducers always return the accumulation of the state (based on all previous and current actions). Therefore, they act as a reducer of state. Each time a Redux reducer is called, the state and action are passed as parameters. This state is then reduced (or accumulated) based on the action, and then the next state is returned. You could *reduce* a collection of actions and an initial state (of the store) on which to perform these actions to get the resulting final state.

[↑ Back to Top](#)

167. How to make AJAX request in Redux?

You can use `redux-thunk` middleware which allows you to define async actions.

Let's take an example of fetching specific account as an AJAX call using *fetch API*:

```
export function fetchAccount(id) {
  return dispatch => {
    dispatch(setLoadingAccountState()) // Show a loading spinner
    fetch(`/account/${id}`, (response) => {
      dispatch(doneFetchingAccount()) // Hide loading spinner
      if (response.status === 200) {
        dispatch(setAccount(response.json)) // Use a normal function to set the re
      } else {
        dispatch(someError)
      }
    })
  }
}

function setAccount(data) {
  return { type: 'SET_Account', data: data }
}
```

[↑ Back to Top](#)

168. Should I keep all component's state in Redux store?

Keep your data in the Redux store, and the UI related state internally in the component.

[↑ Back to Top](#)

169. What is the proper way to access Redux store?

The best way to access your store in a component is to use the `connect()` function, that creates a new component that wraps around your existing one. This pattern is called *Higher-Order Components*, and is generally the preferred way of extending a component's functionality in React. This allows you to map state and action creators to your component, and have them passed in automatically as your store updates.

Let's take an example of `<FilterLink>` component using connect:

```
import { connect } from 'react-redux'
import { setVisibilityFilter } from '../actions'
import Link from '../components/Link'

const mapStateToProps = (state, ownProps) => ({
  active: ownProps.filter === state.visibilityFilter
})

const mapDispatchToProps = (dispatch, ownProps) => ({
  onClick: () => dispatch(setVisibilityFilter(ownProps.filter))
})

const FilterLink = connect(
  mapStateToProps,
  mapDispatchToProps
)(Link)

export default FilterLink
```

Due to it having quite a few performance optimizations and generally being less likely to cause bugs, the Redux developers almost always recommend using `connect()` over accessing the store directly (using context API).

```
class MyComponent {
  someMethod() {
    doSomethingWith(this.context.store)
  }
}
```

 [Back to Top](#)

170. What is the difference between component and container in React Redux?

Component is a class or function component that describes the presentational part of your application.

Container is an informal term for a component that is connected to a Redux store. Containers *subscribe* to Redux state updates and *dispatch* actions, and they usually don't render DOM elements; they delegate rendering to presentational child components.

 [Back to Top](#)

171. What is the purpose of the constants in Redux?

Constants allows you to easily find all usages of that specific functionality across the project when you use an IDE. It also prevents you from introducing silly bugs caused by typos – in which case, you will get a `ReferenceError` immediately.

Normally we will save them in a single file (`constants.js` or `actionTypes.js`).

```
export const ADD_TODO = 'ADD_TODO'
export const DELETE_TODO = 'DELETE_TODO'
export const EDIT_TODO = 'EDIT_TODO'
export const COMPLETE_TODO = 'COMPLETE_TODO'
export const COMPLETE_ALL = 'COMPLETE_ALL'
export const CLEAR_COMPLETED = 'CLEAR_COMPLETED'
```

In Redux, you use them in two places:

i. During action creation:

Let's take `actions.js` :

```
import { ADD_TODO } from './actionTypes';

export function addTodo(text) {
  return { type: ADD_TODO, text }
}
```

ii. In reducers:

Let's create `reducer.js` :

```
import { ADD_TODO } from './actionTypes'

export default (state = [], action) => {
  switch (action.type) {
    case ADD_TODO:
      return [
        ...state,
        {
          text: action.text,
          completed: false
        }
      ];
    default:
      return state
  }
}
```

172. What are the different ways to write `mapDispatchToProps()` ?

There are a few ways of binding *action creators* to `dispatch()` in `mapDispatchToProps()` .

Below are the possible options:

```
const mapDispatchToProps = (dispatch) => ({
  action: () => dispatch(action()))
})
```

```
const mapDispatchToProps = (dispatch) => ({
  action: bindActionCreators(action, dispatch)
})
```

```
const mapDispatchToProps = { action }
```

The third option is just a shorthand for the first one.

 [Back to Top](#)

173. What is the use of the `ownProps` parameter in `mapStateToProps()` and `mapDispatchToProps()` ?

If the `ownProps` parameter is specified, React Redux will pass the props that were passed to the component into your `connect` functions. So, if you use a connected component:

```
import ConnectedComponent from './containers/ConnectedComponent';

<ConnectedComponent user={'john'} />
```

The `ownProps` inside your `mapStateToProps()` and `mapDispatchToProps()` functions will be an object:

```
{ user: 'john' }
```

You can use this object to decide what to return from those functions.

 [Back to Top](#)

174. How to structure Redux top level directories?

Most of the applications has several top-level directories as below:

- i. **Components:** Used for *dumb* components unaware of Redux.

- ii. **Containers**: Used for *smart* components connected to Redux.
- iii. **Actions**: Used for all action creators, where file names correspond to part of the app.
- iv. **Reducers**: Used for all reducers, where files name correspond to state key.
- v. **Store**: Used for store initialization.

This structure works well for small and medium size apps.

 [Back to Top](#)

175. What is redux-saga?

`redux-saga` is a library that aims to make side effects (asynchronous things like data fetching and impure things like accessing the browser cache) in React/Redux applications easier and better.

It is available in NPM:

```
$ npm install --save redux-saga
```

 [Back to Top](#)

176. What is the mental model of redux-saga?

Saga is like a separate thread in your application, that's solely responsible for side effects. `redux-saga` is a redux *middleware*, which means this thread can be started, paused and cancelled from the main application with normal Redux actions, it has access to the full Redux application state and it can dispatch Redux actions as well.

 [Back to Top](#)

177. What are the differences between `call()` and `put()` in redux-saga?

Both `call()` and `put()` are effect creator functions. `call()` function is used to create effect description, which instructs middleware to call the promise. `put()` function creates an effect, which instructs middleware to dispatch an action to the store.

Let's take example of how these effects work for fetching particular user data.

```
function* fetchUserSaga(action) {  
  // `call` function accepts rest arguments, which will be passed to `api.fetchUser`  
  // Instructing middleware to call promise, its resolved value will be assigned to  
  const userData = yield call(api.fetchUser, action.userId)  
  
  // Instructing middleware to dispatch corresponding action.  
}
```

```
yield put({
  type: 'FETCH_USER_SUCCESS',
  userData
})
}
```

 [Back to Top](#)

178. What is Redux Thunk?

Redux Thunk middleware allows you to write action creators that return a function instead of an action. The thunk can be used to delay the dispatch of an action, or to dispatch only if a certain condition is met. The inner function receives the store methods `dispatch()` and `getState()` as parameters.

 [Back to Top](#)

179. What are the differences between `redux-saga` and `redux-thunk` ?

Both *Redux Thunk* and *Redux Saga* take care of dealing with side effects. In most of the scenarios, Thunk uses *Promises* to deal with them, whereas Saga uses *Generators*. Thunk is simple to use and Promises are familiar to many developers, Sagas/Generators are more powerful but you will need to learn them. But both middleware can coexist, so you can start with Thunks and introduce Sagas when/if you need them.

 [Back to Top](#)

180. What is Redux DevTools?

Redux DevTools is a live-editing time travel environment for Redux with hot reloading, action replay, and customizable UI. If you don't want to bother with installing Redux DevTools and integrating it into your project, consider using Redux DevTools Extension for Chrome and Firefox.

 [Back to Top](#)

181. What are the features of Redux DevTools?

Some of the main features of Redux DevTools are below,

- i. Lets you inspect every state and action payload.
- ii. Lets you go back in time by *cancelling* actions.
- iii. If you change the reducer code, each *staged* action will be re-evaluated.
- iv. If the reducers throw, you will see during which action this happened, and what the error was.

- v. With `persistState()` store enhancer, you can persist debug sessions across page reloads.

 [Back to Top](#)

182. What are Redux selectors and why to use them?

Selectors are functions that take Redux state as an argument and return some data to pass to the component.

For example, to get user details from the state:

```
const getUserData = state => state.user.data
```

These selectors have two main benefits,

- i. The selector can compute derived data, allowing Redux to store the minimal possible state
- ii. The selector is not recomputed unless one of its arguments changes

 [Back to Top](#)

183. What is Redux Form?

Redux Form works with React and Redux to enable a form in React to use Redux to store all of its state. Redux Form can be used with raw HTML5 inputs, but it also works very well with common UI frameworks like Material UI, React Widgets and React Bootstrap.

 [Back to Top](#)

184. What are the main features of Redux Form?

Some of the main features of Redux Form are:

- i. Field values persistence via Redux store.
- ii. Validation (sync/async) and submission.
- iii. Formatting, parsing and normalization of field values.

 [Back to Top](#)

185. How to add multiple middlewares to Redux?

You can use `applyMiddleware()`.

For example, you can add `redux-thunk` and `logger` passing them as arguments to `applyMiddleware()`:

```
import { createStore, applyMiddleware } from 'redux'
const createStoreWithMiddleware = applyMiddleware(ReduxThunk, logger)(createStore)
```

[↑ Back to Top](#)

186. How to set initial state in Redux?

You need to pass initial state as second argument to createStore:

```
const rootReducer = combineReducers({
  todos,
  visibilityFilter
})

const initialState = {
  todos: [{ id: 123, name: 'example', completed: false }]
}

const store = createStore(
  rootReducer,
  initialState
)
```

[↑ Back to Top](#)

187. How Relay is different from Redux?

Relay is similar to Redux in that they both use a single store. The main difference is that relay only manages state originated from the server, and all access to the state is used via *GraphQL* queries (for reading data) and mutations (for changing data). Relay caches the data for you and optimizes data fetching for you, by fetching only changed data and nothing more.

188. What is an action in Redux?

Actions are plain JavaScript objects or payloads of information that send data from your application to your store. They are the only source of information for the store. Actions must have a type property that indicates the type of action being performed.

For example, let's take an action which represents adding a new todo item:

```
{
  type: ADD_TODO,
  text: 'Add todo item'
}
```

 [Back to Top](#)

React Native

 [Back to Top](#)

188. What is the difference between React Native and React?

React is a JavaScript library, supporting both front end web and being run on the server, for building user interfaces and web applications.

React Native is a mobile framework that compiles to native app components, allowing you to build native mobile applications (iOS, Android, and Windows) in JavaScript that allows you to use React to build your components, and implements React under the hood.

 [Back to Top](#)

189. How to test React Native apps?

React Native can be tested only in mobile simulators like iOS and Android. You can run the app in your mobile using expo app (<https://expo.io>) Where it syncs using QR code, your mobile and computer should be in same wireless network.

 [Back to Top](#)

190. How to do logging in React Native?

You can use `console.log` , `console.warn` , etc. As of React Native v0.29 you can simply run the following to see logs in the console:

```
$ react-native log-ios  
$ react-native log-android
```

 [Back to Top](#)

191. How to debug your React Native?

Follow the below steps to debug React Native app:

- i. Run your application in the iOS simulator.
- ii. Press `Command + D` and a webpage should open up at `http://localhost:8081/debugger-ui` .
- iii. Enable *Pause On Caught Exceptions* for a better debugging experience.
- iv. Press `Command + Option + I` to open the Chrome Developer tools, or open it via

View -> Developer -> Developer Tools .

v. You should now be able to debug as you normally would.

React supported libraries & Integration

 [Back to Top](#)

192. What is reselect and how it works?

Reselect is a **selector library** (for Redux) which uses *memoization* concept. It was originally written to compute derived data from Redux-like applications state, but it can't be tied to any architecture or library.

Reselect keeps a copy of the last inputs/outputs of the last call, and recomputes the result only if one of the inputs changes. If the same inputs are provided twice in a row, Reselect returns the cached output. Its memoization and cache are fully customizable.

 [Back to Top](#)

193. What is Flow?

Flow is a *static type checker* designed to find type errors in JavaScript. Flow types can express much more fine-grained distinctions than traditional type systems. For example, Flow helps you catch errors involving `null`, unlike most type systems.

 [Back to Top](#)

194. What is the difference between Flow and PropTypes?

Flow is a *static analysis tool* (static checker) which uses a superset of the language, allowing you to add type annotations to all of your code and catch an entire class of bugs at compile time.

PropTypes is a *basic type checker* (runtime checker) which has been patched onto React. It can't check anything other than the types of the props being passed to a given component. If you want more flexible typechecking for your entire project Flow/TypeScript are appropriate choices.

 [Back to Top](#)

195. How to use Font Awesome icons in React?

The below steps followed to include Font Awesome in React:

- i. Install `font-awesome` :

```
$ npm install --save font-awesome
```

ii. Import `font-awesome` in your `index.js` file:

```
import 'font-awesome/css/font-awesome.min.css'
```

iii. Add Font Awesome classes in `className`:

```
render() {
  return <div><i className={'fa fa-spinner'} /></div>
}
```

 [Back to Top](#)

196. What is React Dev Tools?

React Developer Tools let you inspect the component hierarchy, including component props and state. It exists both as a browser extension (for Chrome and Firefox), and as a standalone app (works with other environments including Safari, IE, and React Native).

The official extensions available for different browsers or environments.

- i. Chrome extension
- ii. Firefox extension
- iii. Standalone app (Safari, React Native, etc)

 [Back to Top](#)

197. Why is DevTools not loading in Chrome for local files?

If you opened a local HTML file in your browser (`file:///...`) then you must first open *Chrome Extensions* and check `Allow access to file URLs`.

 [Back to Top](#)

198. How to use Polymer in React?

You need to follow below steps to use Polymer in React,

- i. Create a Polymer element:

```
<link rel='import' href='../bower_components/polymer/polymer.html' />
Polymer({
  is: 'calender-element',
  ready: function() {
```

```
        this.textContent = 'I am a calender'  
    }  
})
```

- ii. Create the Polymer component HTML tag by importing it in a HTML document, e.g.
import it in the `index.html` of your React application:

```
<link rel='import' href='./src/polymer-components/calender-element.html'>
```

- iii. Use that element in the JSX file:

```
import React from 'react'  
  
class MyComponent extends React.Component {  
    render() {  
        return (  
            <calender-element />  
        )  
    }  
}  
  
export default MyComponent
```

 [Back to Top](#)

199. What are the advantages of React over Vue.js?

React has the following advantages over Vue.js:

- i. Gives more flexibility in large apps developing.
- ii. Easier to test.
- iii. Suitable for mobile apps creating.
- iv. More information and solutions available.

Note: The above list of advantages are purely opinionated and it vary based on the professional experience. But they are helpful as base parameters.

 [Back to Top](#)

200. What is the difference between React and Angular?

Let's see the difference between React and Angular in a table format.

| React | Angular |
|-------------------------------------|---|
| React is a library and has only the | Angular is a framework and has complete |

| React | Angular |
|---|--|
| View layer | MVC functionality |
| React handles rendering on the server side | AngularJS renders only on the client side but Angular 2 and above renders on the server side |
| React uses JSX that looks like HTML in JS which can be confusing | Angular follows the template approach for HTML, which makes code shorter and easy to understand |
| React Native, which is a React type to build mobile applications are faster and more stable | Ionic, Angular's mobile native app is relatively less stable and slower |
| In React, data flows only in one way and hence debugging is easy | In Angular, data flows both way i.e it has two-way data binding between children and parent and hence debugging is often difficult |

Note: The above list of differences are purely opinionated and it vary based on the professional experience. But they are helpful as base parameters.

[↑ Back to Top](#)

201. Why React tab is not showing up in DevTools?

When the page loads, *React DevTools* sets a global named `__REACT_DEVTOOLS_GLOBAL_HOOK__`, then React communicates with that hook during initialization. If the website is not using React or if React fails to communicate with DevTools then it won't show up the tab.

[↑ Back to Top](#)

202. What are Styled Components?

`styled-components` is a JavaScript library for styling React applications. It removes the mapping between styles and components, and lets you write actual CSS augmented with JavaScript.

[↑ Back to Top](#)

203. Give an example of Styled Components?

Lets create `<Title>` and `<Wrapper>` components with specific styles for each.

```
import React from 'react'
import styled from 'styled-components'
```

```
// Create a <Title> component that renders an <h1> which is centered, red and size
const Title = styled.h1`
  font-size: 1.5em;
  text-align: center;
  color: palevioletred;
`


// Create a <Wrapper> component that renders a <section> with some padding and a pink background
const Wrapper = styled.section`
  padding: 4em;
  background: papayawhip;
`
```

These two variables, `Title` and `Wrapper`, are now components that you can render just like any other react component.

```
<Wrapper>
  <Title>{'Lets start first styled component!'}</Title>
</Wrapper>
```

 [Back to Top](#)

204. What is Relay?

Relay is a JavaScript framework for providing a data layer and client-server communication to web applications using the React view layer.

 [Back to Top](#)

205. How to use TypeScript in `create-react-app` application?

Starting from `react-scripts@2.1.0` or higher, there is a built-in support for typescript. i.e, `create-react-app` now supports typescript natively. You can just pass `--typescript` option as below

```
npx create-react-app my-app --typescript

# or

yarn create react-app my-app --typescript
```

But for lower versions of react scripts, just supply `--scripts-version` option as `react-scripts-ts` while you create a new project. `react-scripts-ts` is a set of adjustments to take the standard `create-react-app` project pipeline and bring TypeScript into the mix.

Now the project layout should look like the following:

```
my-app/
├── .gitignore
├── images.d.ts
├── node_modules/
├── public/
├── src/
│   └── ...
└── package.json
    ├── tsconfig.json
    ├── tsconfig.prod.json
    ├── tsconfig.test.json
    └── tslint.json
```

Miscellaneous

[↑ Back to Top](#)

206. What are the main features of Reselect library?

Let's see the main features of Reselect library,

- i. Selectors can compute derived data, allowing Redux to store the minimal possible state.
- ii. Selectors are efficient. A selector is not recomputed unless one of its arguments changes.
- iii. Selectors are composable. They can be used as input to other selectors.

207. Give an example of Reselect usage?

Let's take calculations and different amounts of a shipment order with the simplified usage of Reselect:

```
import { createSelector } from 'reselect'

const shopItemsSelector = state => state.shop.items
const taxPercentSelector = state => state.shop.taxPercent

const subtotalSelector = createSelector(
  shopItemsSelector,
  items => items.reduce((acc, item) => acc + item.value, 0)
)

const taxSelector = createSelector(
  subtotalSelector,
  taxPercentSelector,
  (subtotal, taxPercent) => subtotal * (taxPercent / 100)
)
```

```

export const totalSelector = createSelector(
  subtotalSelector,
  taxSelector,
  (subtotal, tax) => ({ total: subtotal + tax })
)

let exampleState = {
  shop: {
    taxPercent: 8,
    items: [
      { name: 'apple', value: 1.20 },
      { name: 'orange', value: 0.95 },
    ]
  }
}

console.log(subtotalSelector(exampleState)) // 2.15
console.log(taxSelector(exampleState)) // 0.172
console.log(totalSelector(exampleState)) // { total: 2.322 }

```

[↑ Back to Top](#)

209. Does the statics object work with ES6 classes in React?

No, `statics` only works with `React.createClass()`:

```

someComponent= React.createClass({
  statics: {
    someMethod: function() {
      // ..
    }
  }
})

```

But you can write statics inside ES6+ classes as below,

```

class Component extends React.Component {
  static propTypes = {
    // ...
  }

  static someMethod() {
    // ...
  }
}

```

or writing them outside class as below,

```

class Component extends React.Component {

```

```
    ....  
}  
  
Component.propTypes = {...}  
Component.someMethod = function(){....}
```

 [Back to Top](#)

210. Can Redux only be used with React?

Redux can be used as a data store for any UI layer. The most common usage is with React and React Native, but there are bindings available for Angular, Angular 2, Vue, Mithril, and more. Redux simply provides a subscription mechanism which can be used by any other code.

 [Back to Top](#)

211. Do you need to have a particular build tool to use Redux?

Redux is originally written in ES6 and transpiled for production into ES5 with Webpack and Babel. You should be able to use it regardless of your JavaScript build process. Redux also offers a UMD build that can be used directly without any build process at all.

 [Back to Top](#)

212. How Redux Form `initialValues` get updated from state?

You need to add `enableReinitialize : true` setting.

```
const InitializeFromStateForm = reduxForm({  
  form: 'initializeFromState',  
  enableReinitialize : true  
})(UserEdit)
```

If your `initialValues` prop gets updated, your form will update too.

 [Back to Top](#)

213. How React PropTypes allow different types for one prop?

You can use `oneOfType()` method of `PropTypes`.

For example, the height property can be defined with either `string` or `number` type as below:

```
Component.PropTypes = {  
  size: PropTypes.oneOfType([
```

```
    PropTypes.string,  
    PropTypes.number  
])  
}
```

[↑ Back to Top](#)

214. Can I import an SVG file as react component?

You can import SVG directly as component instead of loading it as a file. This feature is available with `react-scripts@2.0.0` and higher.

```
import { ReactComponent as Logo } from './logo.svg'  
  
const App = () => (  
  <div>  
    {/* Logo is an actual react component */}  
    <Logo />  
  </div>  
)
```

Note: Don't forget about the curly braces in the import.

[↑ Back to Top](#)

215. Why are inline ref callbacks or functions not recommended?

If the ref callback is defined as an inline function, it will get called twice during updates, first with null and then again with the DOM element. This is because a new instance of the function is created with each render, so React needs to clear the old ref and set up the new one.

```

class UserForm extends Component {
  handleSubmit = () => {
    console.log("Input Value is: ", this.input.value)
  }

  render () {
    return (
      <form onSubmit={this.handleSubmit}>
        <input
          type='text'
          ref={(input) => this.input = input} // Access DOM input in handle submit
          <button type='submit'>Submit</button>
      </form>
    )
  }
}

```

But our expectation is for the ref callback to get called once, when the component mounts. One quick fix is to use the ES7 class property syntax to define the function

```

class UserForm extends Component {
  handleSubmit = () => {
    console.log("Input Value is: ", this.input.value)
  }

  setSearchInput = (input) => {
    this.input = input
  }

  render () {
    return (
      <form onSubmit={this.handleSubmit}>
        <input
          type='text'
          ref={this.setSearchInput} // Access DOM input in handle submit
          <button type='submit'>Submit</button>
      </form>
    )
  }
}

```

****Note:**** In React v16.3,

 [Back to Top](#)

216. What is render hijacking in react?

The concept of render hijacking is the ability to control what a component will output from another component. It actually means that you decorate your component by wrapping it into a Higher-Order component. By wrapping you can inject additional props or make other changes, which can cause changing logic of rendering. It does not actually enable hijacking, but by using HOC you make your component behave in different way.

[↑ Back to Top](#)

217. What are HOC factory implementations?

There are two main ways of implementing HOCs in React.

- i. Props Proxy (PP) and
- ii. Inheritance Inversion (II).

But they follow different approaches for manipulating the *WrappedComponent*.

Props Proxy

In this approach, the render method of the HOC returns a React Element of the type of the WrappedComponent. We also pass through the props that the HOC receives, hence the name **Props Proxy**.

```
function ppHOC(WrappedComponent) {
  return class PP extends React.Component {
    render() {
      return <WrappedComponent {...this.props}/>
    }
  }
}
```

Inheritance Inversion

In this approach, the returned HOC class (Enhancer) extends the WrappedComponent. It is called Inheritance Inversion because instead of the WrappedComponent extending some Enhancer class, it is passively extended by the Enhancer. In this way the relationship between them seems **inverse**.

```
function iiHOC(WrappedComponent) {
  return class Enhancer extends WrappedComponent {
    render() {
      return super.render()
    }
  }
}
```

 [Back to Top](#)

218. How to pass numbers to React component?

You should be passing the numbers via curly braces({}) where as strings in quotes

```
React.render(<User age={30} department="IT" />, document.getElementById('cont
```

 [Back to Top](#)

219. Do I need to keep all my state into Redux? Should I ever use react internal state?

It is up to developer decision. i.e, It is developer job to determine what kinds of state make up your application, and where each piece of state should live. Some users prefer to keep every single piece of data in Redux, to maintain a fully serializable and controlled version of their application at all times. Others prefer to keep non-critical or UI state, such as "is this dropdown currently open", inside a component's internal state.

Below are the thumb rules to determine what kind of data should be put into Redux

- i. Do other parts of the application care about this data?
- ii. Do you need to be able to create further derived data based on this original data?
- iii. Is the same data being used to drive multiple components?
- iv. Is there value to you in being able to restore this state to a given point in time (ie, time travel debugging)?
- v. Do you want to cache the data (i.e, use what's in state if it's already there instead of re-requesting it)?

 [Back to Top](#)

220. What is the purpose of registerServiceWorker in React?

React creates a service worker for you without any configuration by default. The service worker is a web API that helps you cache your assets and other files so that when the user is offline or on slow network, he/she can still see results on the screen, as such, it helps you build a better user experience, that's what you should know about service worker's for now. It's all about adding offline capabilities to your site.

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
import registerServiceWorker from './registerServiceWorker';

ReactDOM.render(<App />, document.getElementById('root'));
```

```
registerServiceWorker();
```

[↑ Back to Top](#)

221. What is React memo function?

Class components can be restricted from rendering when their input props are the same using **PureComponent** or **shouldComponentUpdate**. Now you can do the same with function components by wrapping them in **React.memo**.

```
const MyComponent = React.memo(function MyComponent(props) {  
  /* only rerenders if props change */  
});
```

[↑ Back to Top](#)

222. What is React lazy function?

The **React.lazy** function lets you render a dynamic import as a regular component. It will automatically load the bundle containing the **OtherComponent** when the component gets rendered. This must return a Promise which resolves to a module with a default export containing a React component.

```
const OtherComponent = React.lazy(() => import('./OtherComponent'));  
  
function MyComponent() {  
  return (  
    <div>  
      <OtherComponent />  
    </div>  
  );  
}
```

Note: **React.lazy** and **Suspense** is not yet available for server-side rendering. If you want to do code-splitting in a server rendered app, we still recommend **React Loadable**.

[↑ Back to Top](#)

223. How to prevent unnecessary updates using setState?

You can compare current value of the state with an existing state value and decide whether to rerender the page or not. If the values are same then you need to return **null** to stop re-rendering otherwise return the latest state value.

For example, the user profile information is conditionally rendered as follows,

```
getUserProfile = user => {
```

```

const latestAddress = user.address;
this.setState(state => {
  if (state.address === latestAddress) {
    return null;
  } else {
    return { title: latestAddress };
  }
});
};

```

[!\[\]\(e0e6e2e1424dedb544eaa802ce55752a_img.jpg\) Back to Top](#)

224. How do you render Array, Strings and Numbers in React 16 Version?

Arrays: Unlike older releases, you don't need to make sure `render` method return a single element in React16. You are able to return multiple sibling elements without a wrapping element by returning an array.

For example, let us take the below list of developers,

```

const ReactJSDevs = () => {
  return [
    <li key="1">John</li>,
    <li key="2">Jackie</li>,
    <li key="3">Jordan</li>
  ];
}

```

You can also merge this array of items in another array component.

```

const JSDevs = () => {
  return (
    <ul>
      <li>Brad</li>
      <li>Brodge</li>
      <ReactJSDevs/>
      <li>Brandon</li>
    </ul>
  );
}

```

Strings and Numbers: You can also return string and number type from the `render` method.

```

render() {
  return 'Welcome to ReactJS questions';
}
// Number

```

```
render() {
  return 2018;
}
```

[↑ Back to Top](#)

225. How to use class field declarations syntax in React classes?

React Class Components can be made much more concise using the class field declarations. You can initialize local state without using the constructor and declare class methods by using arrow functions without the extra need to bind them.

Let's take a counter example to demonstrate class field declarations for state without using constructor and methods without binding,

```
class Counter extends Component {
  state = { value: 0 };

  handleIncrement = () => {
    this.setState(prevState => ({
      value: prevState.value + 1
    }));
  };

  handleDecrement = () => {
    this.setState(prevState => ({
      value: prevState.value - 1
    }));
  };

  render() {
    return (
      <div>
        {this.state.value}

        <button onClick={this.handleIncrement}>+</button>
        <button onClick={this.handleDecrement}>-</button>
      </div>
    )
  }
}
```

[↑ Back to Top](#)

226. What are hooks?

Hooks is a new feature(React 16.8) that lets you use state and other React features without writing a class.

Let's see an example of useState hook example,

```
import { useState } from 'react';

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

 [Back to Top](#)

227. What are the rules needs to follow for hooks?

You need to follow two rules in order to use hooks,

- i. Call Hooks only at the top level of your react functions. i.e, You shouldn't call Hooks inside loops, conditions, or nested functions. This will ensure that Hooks are called in the same order each time a component renders and it preserves the state of Hooks between multiple useState and useEffect calls.
- ii. Call Hooks from React Functions only. i.e, You shouldn't call Hooks from regular JavaScript functions.

 [Back to Top](#)

228. How to ensure hooks followed the rules in your project?

React team released an ESLint plugin called **eslint-plugin-react-hooks** that enforces these two rules. You can add this plugin to your project using the below command,

```
npm install eslint-plugin-react-hooks@next
```

And apply the below config in your ESLint config file,

```
// Your ESLint configuration
{
  "plugins": [
    // ...
    "react-hooks"
  ],
  "rules": {
```

```

    // ...
    "react-hooks/rules-of-hooks": "error"
}
}

```

Note: This plugin is intended to use in Create React App by default.

[↑ Back to Top](#)

229. What are the differences between Flux and Redux?

Below are the major differences between Flux and Redux

| Flux | Redux |
|--|--|
| State is mutable | State is immutable |
| The Store contains both state and change logic | The Store and change logic are separate |
| There are multiple stores exist | There is only one store exist |
| All the stores are disconnected and flat | Single store with hierarchical reducers |
| It has a singleton dispatcher | There is no concept of dispatcher |
| React components subscribe to the store | Container components uses connect function |

[↑ Back to Top](#)

230. What are the benefits of React Router V4?

Below are the main benefits of React Router V4 module,

- i. In React Router v4(version 4), the API is completely about components. A router can be visualized as a single component(`<BrowserRouter>`) which wraps specific child router components(`<Route>`).
- ii. You don't need to manually set history. The router module will take care history by wrapping routes with `<BrowserRouter>` component.
- iii. The application size is reduced by adding only the specific router module(Web, core, or native)

[↑ Back to Top](#)

231. Can you describe about componentDidCatch lifecycle method signature?

The **componentDidCatch** lifecycle method is invoked after an error has been thrown by a descendant component. The method receives two parameters,

- i. error: - The error object which was thrown
- ii. info: - An object with a componentStack key contains the information about which component threw the error.

The method structure would be as follows

```
componentDidCatch(error, info)
```

 [Back to Top](#)

232. In which scenarios error boundaries do not catch errors?

Below are the cases in which error boundaries doesn't work,

- i. Inside Event handlers
- ii. Asynchronous code using **setTimeout** or **requestAnimationFrame** callbacks
- iii. During Server side rendering
- iv. When errors thrown in the error boundary code itself

 [Back to Top](#)

233. Why do not you need error boundaries for event handlers?

Error boundaries do not catch errors inside event handlers. Event handlers don't happen or invoked during rendering time unlike render method or lifecycle methods. So React knows how to recover these kind of errors in event handlers. If still you need to catch an error inside event handler, use the regular JavaScript try / catch statement as below

```
class MyComponent extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = { error: null };  
  }  
  
  handleClick = () => {  
    try {  
      // Do something that could throw  
    } catch (error) {  
      this.setState({ error });  
    }  
  }  
  
  render() {
```

```
if (this.state.error) {
  return <h1>Caught an error.</h1>
}
return <div onClick={this.handleClick}>Click Me</div>
}
```

The above code is catching the error using vanilla javascript try/catch block instead of error boundaries.

[↑ Back to Top](#)

234. What is the difference between try catch block and error boundaries?

Try catch block works with imperative code whereas error boundaries are meant for declarative code to render on the screen.

For example, the try catch block used for below imperative code

```
try {
  showButton();
} catch (error) {
  // ...
}
```

Whereas error boundaries wrap declarative code as below,

```
<ErrorBoundary>
  <MyComponent />
</ErrorBoundary>
```

So if an error occurs in a `componentDidUpdate` method caused by a `setState` somewhere deep in the tree, it will still correctly propagate to the closest error boundary.

[↑ Back to Top](#)

235. What is the behavior of uncaught errors in react 16?

In React 16, errors that were not caught by any error boundary will result in unmounting of the whole React component tree. The reason behind this decision is that it is worse to leave corrupted UI in place than to completely remove it. For example, it is worse for a payments app to display a wrong amount than to render nothing.

[↑ Back to Top](#)

236. What is the proper placement for error boundaries?

The granularity of error boundaries usage is up to the developer based on project needs. You can follow either of these approaches,

- i. You can wrap top-level route components to display a generic error message for the entire application.
- ii. You can also wrap individual components in an error boundary to protect them from crashing the rest of the application.

[↑ Back to Top](#)

237. What is the benefit of component stack trace from error boundary?

Apart from error messages and javascript stack, React16 will display the component stack trace with file names and line numbers using error boundary concept.

For example, BuggyCounter component displays the component stack trace as below,

```
▶ React caught an error thrown by BuggyCounter. You should fix this error in your code. react-dom.development.js:7708
React will try to recreate this component tree from scratch using the error boundary you provided, ErrorBoundary.

Error: I crashed!

The error is located at:
  in BuggyCounter (at App.js:26)
  in ErrorBoundary (at App.js:21)
  in div (at App.js:8)
  in App (at index.js:5)
```

[↑ Back to Top](#)

238. What is the required method to be defined for a class component?

The `render()` method is the only required method in a class component. i.e, All methods other than render method are optional for a class component.

[↑ Back to Top](#)

239. What are the possible return types of render method?

Below are the list of following types used and return from render method,

- i. **React elements:** Elements that instruct React to render a DOM node. It includes html elements such as `<div/>` and user defined elements.
- ii. **Arrays and fragments:** Return multiple elements to render as Arrays and Fragments to wrap multiple elements
- iii. **Portals:** Render children into a different DOM subtree.
- iv. **String and numbers:** Render both Strings and Numbers as text nodes in the DOM
- v. **Booleans or null:** Doesn't render anything but these types are used to conditionally render content.

[↑ Back to Top](#)

240. What is the main purpose of constructor?

The constructor is mainly used for two purposes,

- i. To initialize local state by assigning object to this.state
- ii. For binding event handler methods to the instance For example, the below code covers both the above cases,

```
constructor(props) {  
  super(props);  
  // Don't call this.setState() here!  
  this.state = { counter: 0 };  
  this.handleClick = this.handleClick.bind(this);  
}
```

 [Back to Top](#)

241. Is it mandatory to define constructor for React component?

No, it is not mandatory. i.e, If you don't initialize state and you don't bind methods, you don't need to implement a constructor for your React component.

 [Back to Top](#)

242. What are default props?

The defaultProps are defined as a property on the component class to set the default props for the class. This is used for undefined props, but not for null props.

For example, let us create color default prop for the button component,

```
class MyButton extends React.Component {  
  // ...  
}  
  
MyButton.defaultProps = {  
  color: 'red'  
};
```

If props.color is not provided then it will set the default value to 'red'. i.e, Whenever you try to access the color prop it uses default value

```
render() {  
  return <MyButton /> ; // props.color will be set to red  
}
```

Note: If you provide null value then it remains null value.

[↑ Back to Top](#)

243. Why should not call setState in componentWillMount?

You should not call `setState()` in `componentWillUnmount()` because once a component instance is unmounted, it will never be mounted again.

[↑ Back to Top](#)

244. What is the purpose of getDerivedStateFromError?

This lifecycle method is invoked after an error has been thrown by a descendant component. It receives the error that was thrown as a parameter and should return a value to update state.

The signature of the lifecycle method is as follows,

```
static getDerivedStateFromError(error)
```

Let us take error boundary use case with the above lifecycle method for demonstration purpose,

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // Update state so the next render will show the fallback UI.
    return { hasError: true };
  }

  render() {
    if (this.state.hasError) {
      // You can render any custom fallback UI
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
  }
}
```

[↑ Back to Top](#)

245. What is the methods order when component re-rendered?

An update can be caused by changes to props or state. The below methods are called in the following order when a component is being re-rendered.

- i. static getDerivedStateFromProps()
- ii. shouldComponentUpdate()
- iii. render()
- iv. getSnapshotBeforeUpdate()
- v. componentDidUpdate()

[↑ Back to Top](#)

246. What are the methods invoked during error handling?

Below methods are called when there is an error during rendering, in a lifecycle method, or in the constructor of any child component.

- i. static getDerivedStateFromError()
- ii. componentDidCatch()

[↑ Back to Top](#)

247. What is the purpose of displayName class property?

The displayName string is used in debugging messages. Usually, you don't need to set it explicitly because it's inferred from the name of the function or class that defines the component. You might want to set it explicitly if you want to display a different name for debugging purposes or when you create a higher-order component.

For example, To ease debugging, choose a display name that communicates that it's the result of a withSubscription HOC.

```
function withSubscription(WrappedComponent) {  
  class WithSubscription extends React.Component {/* ... */}  
  WithSubscription.displayName = `WithSubscription(${getDisplayName(WrappedComponent)})`;  
  return WithSubscription;  
}  
function getDisplayName(WrappedComponent) {  
  return WrappedComponent.displayName || WrappedComponent.name || 'Component';  
}
```

[↑ Back to Top](#)

248. What is the browser support for react applications?

React supports all popular browsers, including Internet Explorer 9 and above, although some polyfills are required for older browsers such as IE 9 and IE 10. If you use **es5-shim**

and es5-sham polyfill then it even support old browsers that doesn't support ES5 methods.

[↑ Back to Top](#)

249. What is the purpose of `unmountComponentAtNode` method?

This method is available from react-dom package and it removes a mounted React component from the DOM and clean up its event handlers and state. If no component was mounted in the container, calling this function does nothing. Returns true if a component was unmounted and false if there was no component to unmount.

The method signature would be as follows,

```
ReactDOM.unmountComponentAtNode(container)
```

[↑ Back to Top](#)

250. What is code-splitting?

Code-Splitting is a feature supported by bundlers like Webpack and Browserify which can create multiple bundles that can be dynamically loaded at runtime. The react project supports code splitting via dynamic import() feature.

For example, in the below code snippets, it will make moduleA.js and all its unique dependencies as a separate chunk that only loads after the user clicks the 'Load' button.
moduleA.js

```
const moduleA = 'Hello';

export { moduleA };
```

App.js

```
import React, { Component } from 'react';

class App extends Component {
  handleClick = () => {
    import('./moduleA')
      .then(({ moduleA }) => {
        // Use moduleA
      })
      .catch(err => {
        // Handle failure
      });
  };
}
```

```

    render() {
      return (
        <div>
          <button onClick={this.handleClick}>Load</button>
        </div>
      );
    }
  }

export default App;

```

[↑ Back to Top](#)

251. What is the benefit of strict mode?

The will be helpful in the below cases

- i. Identifying components with **unsafe lifecycle methods**.
- ii. Warning about **legacy string ref API** usage.
- iii. Detecting unexpected **side effects**.
- iv. Detecting **legacy context API**.
- v. Warning about deprecated `findDOMNode` usage

[↑ Back to Top](#)

252. What are Keyed Fragments?

The Fragments declared with the explicit `<React.Fragment>` syntax may have keys. The general use case is mapping a collection to an array of fragments as below,

```

function Glossary(props) {
  return (
    <dl>
      {props.items.map(item => (
        // Without the `key`, React will fire a key warning
        <React.Fragment key={item.id}>
          <dt>{item.term}</dt>
          <dd>{item.description}</dd>
        </React.Fragment>
      ))}
    </dl>
  );
}

```

Note: `key` is the only attribute that can be passed to Fragment. In the future, there might be a support for additional attributes, such as event handlers.

[↑ Back to Top](#)

253. Does React support all HTML attributes?

As of React 16, both standard or custom DOM attributes are fully supported. Since React components often take both custom and DOM-related props, React uses the camelCase convention just like the DOM APIs.

Let us take few props with respect to standard HTML attributes,

```
<div tabIndex="-1" />      // Just like node.tabIndex DOM API  
<div className="Button" /> // Just like node.className DOM API  
<input readOnly={true} /> // Just like node.readOnly DOM API
```

These props work similarly to the corresponding HTML attributes, with the exception of the special cases. It also support all SVG attributes.

 [Back to Top](#)

254. What are the limitations with HOCs?

Higher-order components come with a few caveats apart from its benefits. Below are the few listed in an order,

- i. **Don't use HOCs inside the render method:** It is not recommended to apply a HOC to a component within the render method of a component.

```
render() {  
  // A new version of EnhancedComponent is created on every render  
  // EnhancedComponent1 !== EnhancedComponent2  
  const EnhancedComponent = enhance(MyComponent);  
  // That causes the entire subtree to unmount/remount each time!  
  return <EnhancedComponent />;  
}
```

The above code impact performance by remounting a component that causes the state of that component and all of its children to be lost. Instead, apply HOCs outside the component definition so that the resulting component is created only once.

- ii. **Static methods must be copied over:** When you apply a HOC to a component the new component does not have any of the static methods of the original component

```
// Define a static method  
WrappedComponent.staticMethod = function() {/*...*/}  
// Now apply a HOC  
const EnhancedComponent = enhance(WrappedComponent);  
  
// The enhanced component has no static method
```

```
typeof EnhancedComponent.staticMethod === 'undefined' // true
```

You can overcome this by copying the methods onto the container before returning it,

```
function enhance(WrappedComponent) {
  class Enhance extends React.Component {/*...*/}
  // Must know exactly which method(s) to copy :(
  Enhance.staticMethod = WrappedComponent.staticMethod;
  return Enhance;
}
```

iii. **Refs aren't passed through:** For HOCs you need to pass through all props to the wrapped component but this does not work for refs. This is because ref is not really a prop similar to key. In this case you need to use the `React.forwardRef` API

 [Back to Top](#)

255. How to debug forwardRefs in DevTools?

`React.forwardRef` accepts a render function as parameter and DevTools uses this function to determine what to display for the ref forwarding component.

For example, If you don't name the render function or not using `displayName` property then it will appear as "ForwardRef" in the DevTools,

```
const WrappedComponent = React.forwardRef((props, ref) => {
  return <LogProps {...props} forwardedRef={ref} />;
});
```

But If you name the render function then it will appear as "ForwardRef(`myFunction`)"

```
const WrappedComponent = React.forwardRef(
  function myFunction(props, ref) {
    return <LogProps {...props} forwardedRef={ref} />;
  }
);
```

As an alternative, You can also set `displayName` property for `forwardRef` function,

```
function logProps(Component) {
  class LogProps extends React.Component {
    // ...
  }

  function forwardRef(props, ref) {
```

```

    return <LogProps {...props} forwardedRef={ref} />;
}

// Give this component a more helpful display name in DevTools.
// e.g. "ForwardRef(logProps(MyComponent))"
const name = Component.displayName || Component.name;
forwardRef.displayName = `logProps(${name})`;

return React.forwardRef(forwardRef);
}

```

[↑ Back to Top](#)

256. When component props defaults to true?

If you pass no value for a prop, it defaults to true. This behavior is available so that it matches the behavior of HTML.

For example, below expressions are equivalent,

```

<MyInput autocomplete />

<MyInput autocomplete={true} />

```

Note: It is not recommended to use this approach because it can be confused with the ES6 object shorthand (example, `{name}` which is short for `{name: name}`)

[↑ Back to Top](#)

257. What is NextJS and major features of it?

Next.js is a popular and lightweight framework for static and server-rendered applications built with React. It also provides styling and routing solutions. Below are the major features provided by NextJS,

- i. Server-rendered by default
- ii. Automatic code splitting for faster page loads
- iii. Simple client-side routing (page based)
- iv. Webpack-based dev environment which supports (HMR)
- v. Able to implement with Express or any other Node.js HTTP server
- vi. Customizable with your own Babel and Webpack configurations

[↑ Back to Top](#)

258. How do you pass an event handler to a component?

You can pass event handlers and other functions as props to child components. It can be used in child component as below,

```
<button onClick={this.handleClick}>
```

 [Back to Top](#)

259. Is it good to use arrow functions in render methods?

Yes, You can use. It is often the easiest way to pass parameters to callback functions. But you need to optimize the performance while using it.

```
class Foo extends Component {  
  handleClick() {  
    console.log('Click happened');  
  }  
  render() {  
    return <button onClick={() => this.handleClick()}>Click Me</button>;  
  }  
}
```

Note: Using an arrow function in render method creates a new function each time the component renders, which may have performance implications

 [Back to Top](#)

260. How to prevent a function from being called multiple times?

If you use an event handler such as **onClick** or **onScroll** and want to prevent the callback from being fired too quickly, then you can limit the rate at which callback is executed. This can be achieved in the below possible ways,

- i. **Throttling:** Changes based on a time based frequency. For example, it can be used using `_throttle` lodash function
- ii. **Debouncing:** Publish changes after a period of inactivity. For example, it can be used using `_debounce` lodash function
- iii. **RequestAnimationFrame throttling:** Changes based on `requestAnimationFrame`. For example, it can be used using `raf-schd` lodash function

 [Back to Top](#)

261. How JSX prevents Injection Attacks?

React DOM escapes any values embedded in JSX before rendering them. Thus it ensures that you can never inject anything that's not explicitly written in your application. Everything is converted to a string before being rendered.

For example, you can embed user input as below,

```
const name = response.potentiallyMaliciousInput;
const element = <h1>{name}</h1>;
```

This way you can prevent XSS(Cross-site-scripting) attacks in the application.

[↑ Back to Top](#)

262. How do you update rendered elements?

You can update UI(represented by rendered element) by passing the newly created element to ReactDOM's render method.

For example, lets take a ticking clock example, where it updates the time by calling render method multiple times,

```
function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  ReactDOM.render(element, document.getElementById('root'));
}

setInterval(tick, 1000);
```

[↑ Back to Top](#)

263. How do you say that props are read only?

When you declare a component as a function or a class, it must never modify its own props.

Let us take a below capital function,

```
function capital(amount, interest) {
  return amount + interest;
}
```

The above function is called "pure" because it does not attempt to change their inputs, and always return the same result for the same inputs. Hence, React has a single rule saying "All React components must act like pure functions with respect to their props."

[↑ Back to Top](#)

264. How do you say that state updates are merged?

When you call `setState()` in the component, React merges the object you provide into the current state.

For example, let us take a facebook user with posts and comments details as state variables,

```
constructor(props) {
  super(props);
  this.state = {
    posts: [],
    comments: []
  };
}
```

Now you can update them independently with separate `setState()` calls as below,

```
componentDidMount() {
  fetchPosts().then(response => {
    this.setState({
      posts: response.posts
    });
  });

  fetchComments().then(response => {
    this.setState({
      comments: response.comments
    });
  });
}
```

As mentioned in the above code snippets, `this.setState({comments})` updates only comments variable without modifying or replacing posts variable.

[↑ Back to Top](#)

265. How do you pass arguments to an event handler?

During iterations or loops, it is common to pass an extra parameter to an event handler. This can be achieved through arrow functions or bind method.

Let us take an example of user details updated in a grid,

```
<button onClick={(e) => this.updateUser(userId, e)}>Update User details</button>
<button onClick={this.updateUser.bind(this, userId)}>Update User details</button>
```

In both the approaches, the synthetic argument e is passed as a second argument. You need to pass it explicitly for arrow functions and it forwarded automatically for bind method.

[↑ Back to Top](#)

266. How to prevent component from rendering?

You can prevent component from rendering by returning null based on specific condition. This way it can conditionally render component.

```
function Greeting(props) {
  if (!props.loggedIn) {
    return null;
  }

  return (
    <div className="greeting">
      welcome, {props.name}
    </div>
  );
}

class User extends React.Component {
  constructor(props) {
    super(props);
    this.state = {loggedIn: false, name: 'John'};
  }

  render() {
    return (
      <div>
        //Prevent component render if it is not loggedIn
        <Greeting loggedIn={this.state.loggedIn} />
        <UserDetails name={this.state.name}>
      </div>
    );
  }
}
```

In the above example, the greeting component skips its rendering section by applying condition and returning null value.

[↑ Back to Top](#)

267. What are the conditions to safely use the index as a key?

There are three conditions to make sure, it is safe use the index as a key.

- i. The list and items are static— they are not computed and do not change
- ii. The items in the list have no ids
- iii. The list is never reordered or filtered.

[↑ Back to Top](#)

268. Is it keys should be globally unique?

Keys used within arrays should be unique among their siblings but they don't need to be globally unique. i.e, You can use the same keys with two different arrays.

For example, the below book component uses two arrays with different arrays,

```
function Book(props) {
  const index = (
    <ul>
      {props.pages.map((page) =>
        <li key={page.id}>
          {page.title}
        </li>
      )}
    </ul>
  );
  const content = props.pages.map((page) =>
    <div key={page.id}>
      <h3>{page.title}</h3>
      <p>{page.content}</p>
      <p>{page.pageNumber}</p>
    </div>
  );
  return (
    <div>
      {index}
      <hr />
      {content}
    </div>
  );
}
```

[↑ Back to Top](#)

269. What is the popular choice for form handling?

Formik is a form library for react which provides solutions such as validation, keeping track of the visited fields, and handling form submission.

In detail, You can categorize them as follows,

- i. Getting values in and out of form state

ii. Validation and error messages

iii. Handling form submission

It is used to create a scalable, performant, form helper with a minimal API to solve annoying stuff.

 [Back to Top](#)

270. What are the advantages of formik over redux form library?

Below are the main reasons to recommend formik over redux form library,

- i. The form state is inherently short-term and local, so tracking it in Redux (or any kind of Flux library) is unnecessary.
- ii. Redux-Form calls your entire top-level Redux reducer multiple times ON EVERY SINGLE KEYSTROKE. This way it increases input latency for large apps.
- iii. Redux-Form is 22.5 kB minified gzipped whereas Formik is 12.7 kB

 [Back to Top](#)

271. Why do you not required to use inheritance?

In React, it is recommended using composition instead of inheritance to reuse code between components. Both Props and composition give you all the flexibility you need to customize a component's look and behavior in an explicit and safe way. Whereas, If you want to reuse non-UI functionality between components, it is suggested to extracting it into a separate JavaScript module. Later components import it and use that function, object, or a class, without extending it.

 [Back to Top](#)

272. Can I use web components in react application?

Yes, you can use web components in a react application. Even though many developers won't use this combination, it may require especially if you are using third-party UI components that are written using Web Components.

For example, let us use Vaadin date picker web component as below,

```
import React, { Component } from 'react';
import './App.css';
import '@vaadin/vaadin-date-picker';
class App extends Component {
  render() {
    return (
      <div className="App">
        <vaadin-date-picker label="When were you born?"></vaadin-date-picker>
      </div>
    );
  }
}
```

```
        </div>
    );
}
}

export default App;
```

[↑ Back to Top](#)

273. What is dynamic import?

The dynamic import() syntax is a ECMAScript proposal not currently part of the language standard. It is expected to be accepted in the near future. You can achieve code-splitting into your app using dynamic import.

Let's take an example of addition,

i. Normal Import

```
import { add } from './math';
console.log(add(10, 20));
```

ii. Dynamic Import

```
import("./math").then(math => {
  console.log(math.add(10, 20));
});
```

[↑ Back to Top](#)

274. What are loadable components?

If you want to do code-splitting in a server rendered app, it is recommend to use Loadable Components because React.lazy and Suspense is not yet available for server-side rendering. Loadable lets you render a dynamic import as a regular component.

Lets take an example,

```
import loadable from '@loadable/component'

const OtherComponent = loadable(() => import('./OtherComponent'))

function MyComponent() {
  return (
    <div>
      <OtherComponent />
    </div>
  )
}
```

Now OtherComponent will be loaded in a separated bundle

[↑ Back to Top](#)

275. What is suspense component?

If the module containing the dynamic import is not yet loaded by the time parent component renders, you must show some fallback content while you're waiting for it to load using a loading indicator. This can be done using **Suspense** component.

For example, the below code uses suspense component,

```
const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <OtherComponent />
      </Suspense>
    </div>
  );
}
```

As mentioned in the above code, Suspense is wrapped above the lazy component.

[↑ Back to Top](#)

276. What is route based code splitting?

One of the best place to do code splitting is with routes. The entire page is going to re-render at once so users are unlikely to interact with other elements in the page at the same time. Due to this, the user experience won't be disturbed.

Let us take an example of route based website using libraries like React Router with React.lazy,

```
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';
import React, { Suspense, lazy } from 'react';

const Home = lazy(() => import('./routes/Home'));
const About = lazy(() => import('./routes/About'));

const App = () => (
  <Router>
    <Suspense fallback={<div>Loading...</div>}>
      <Switch>
        <Route exact path="/" component={Home}/>
        <Route path="/about" component={About}/>
      </Switch>
    </Suspense>
  </Router>
)
```

```
        </Switch>
    </Suspense>
</Router>
);
```

In the above code, the code splitting will happen at each route level.

[↑ Back to Top](#)

277. Give an example on How to use context?

Context is designed to share data that can be considered **global** for a tree of React components.

For example, in the code below lets manually thread through a "theme" prop in order to style the Button component.

```
// Lets create a context with a default theme value "luna"
const ThemeContext = React.createContext('luna');
// Create App component where it uses provider to pass theme value in the tree
class App extends React.Component {
  render() {
    return (
      <ThemeContext.Provider value="nova">
        <Toolbar />
      </ThemeContext.Provider>
    );
  }
}
// A middle component where you don't need to pass theme prop anymore
function Toolbar(props) {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}
// Lets read theme value in the button component to use
class ThemedButton extends React.Component {
  static contextType = ThemeContext;
  render() {
    return <Button theme={this.context} />;
  }
}
```

[↑ Back to Top](#)

278. What is the purpose of default value in context?

The defaultValue argument is only used when a component does not have a matching Provider above it in the tree. This can be helpful for testing components in isolation without wrapping them.

Below code snippet provides default theme value as Luna.

```
const MyContext = React.createContext(defaultValue);
```

 [Back to Top](#)

279. How do you use contextType?

ContextType is used to consume the context object. The contextType property can be used in two ways,

- i. **contextType as property of class:** The contextType property on a class can be assigned a Context object created by React.createContext(). After that, you can consume the nearest current value of that Context type using this.context in any of the lifecycle methods and render function.

Lets assign contextType property on MyClass as below,

```
class MyClass extends React.Component {  
  componentDidMount() {  
    let value = this.context;  
    /* perform a side-effect at mount using the value of MyContext */  
  }  
  componentDidUpdate() {  
    let value = this.context;  
    /* ... */  
  }  
  componentWillUnmount() {  
    let value = this.context;  
    /* ... */  
  }  
  render() {  
    let value = this.context;  
    /* render something based on the value of MyContext */  
  }  
}  
MyClass.contextType = MyContext;
```

- ii. **Static field** You can use a static class field to initialize your contextType using public class field syntax.

```
class MyClass extends React.Component {  
  static contextType = MyContext;  
  render() {
```

```
    let value = this.context;
    /* render something based on the value */
}
}
```

[↑ Back to Top](#)

280. What is a consumer?

A Consumer is a React component that subscribes to context changes. It requires a function as a child which receives current context value as argument and returns a react node. The value argument passed to the function will be equal to the value prop of the closest Provider for this context above in the tree.

Lets take a simple example,

```
<MyContext.Consumer>
  {value => /* render something based on the context value */}
</MyContext.Consumer>
```

[↑ Back to Top](#)

281. How do you solve performance corner cases while using context?

The context uses reference identity to determine when to re-render, there are some gotchas that could trigger unintentional renders in consumers when a provider's parent re-renders.

For example, the code below will re-render all consumers every time the Provider re-renders because a new object is always created for value.

```
class App extends React.Component {
  render() {
    return (
      <Provider value={{something: 'something'}}>
        <Toolbar />
      </Provider>
    );
  }
}
```

This can be solved by lifting up the value to parent state,

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
```

```

        value: {something: 'something'},
    );
}

render() {
    return (
        <Provider value={this.state.value}>
            <Toolbar />
        </Provider>
    );
}
}

```

 [Back to Top](#)

282. What is the purpose of forward ref in HOCs?

Refs will not get passed through because ref is not a prop. It handled differently by React just like **key**. If you add a ref to a HOC, the ref will refer to the outermost container component, not the wrapped component. In this case, you can use Forward Ref API. For example, we can explicitly forward refs to the inner FancyButton component using the `React.forwardRef` API.

The below HOC logs all props,

```

```javascript
function logProps(Component) {
 class LogProps extends React.Component {
 componentDidUpdate(prevProps) {
 console.log('old props:', prevProps);
 console.log('new props:', this.props);
 }

 render() {
 const {forwardedRef, ...rest} = this.props;

 // Assign the custom prop "forwardedRef" as a ref
 return <Component ref={forwardedRef} {...rest} />;
 }
 }

 return React.forwardRef((props, ref) => {
 return <LogProps {...props} forwardedRef={ref} />;
 });
}
```

```

Let's use this HOC to log all props that get passed to our "fancy button" component,

```
```javascript
class FancyButton extends React.Component {
 focus() {
 // ...
 }

 // ...
}

export default logProps(FancyButton);
```
```

Now lets create a ref and pass it to FancyButton component. In this case, you can set focus to button element.

```
```javascript
import FancyButton from './FancyButton';

const ref = React.createRef();
ref.current.focus();

<FancyButton
 label="Click Me"
 handleClick={handleClick}
 ref={ref}
/>;
```
```

[↑ Back to Top](#)

283. Is it ref argument available for all functions or class components?

Regular function or class components don't receive the ref argument, and ref is not available in props either. The second ref argument only exists when you define a component with `React.forwardRef` call.

[↑ Back to Top](#)

284. Why do you need additional care for component libraries while using forward refs?

When you start using `forwardRef` in a component library, you should treat it as a breaking change and release a new major version of your library. This is because your library likely has a different behavior such as what refs get assigned to, and what types are exported. These changes can break apps and other libraries that depend on the old behavior.

[↑ Back to Top](#)

285. How to create react class components without ES6?

If you don't use ES6 then you may need to use the `create-react-class` module instead. For default props, you need to define `getDefaultProps()` as a function on the passed object. Whereas for initial state, you have to provide a separate `getInitialState` method that returns the initial state.

```
var Greeting = createReactClass({
  getDefaultProps: function() {
    return {
      name: 'Jhohn'
    };
  },
  getInitialState: function() {
    return {message: this.props.message};
  },
  handleClick: function() {
    console.log(this.state.message);
  },
  render: function() {
    return <h1>Hello, {this.props.name}</h1>;
  }
});
```

Note: If you use `createReactClass` then auto binding is available for all methods. i.e, You don't need to use `.bind(this)` with in constructor for event handlers.

 [Back to Top](#)

286. Is it possible to use react without JSX?

Yes, JSX is not mandatory for using React. Actually it is convenient when you don't want to set up compilation in your build environment. Each JSX element is just syntactic sugar for calling `React.createElement(component, props, ...children)`.

For example, let us take a greeting example with JSX,

```
class Greeting extends React.Component {
  render() {
    return <div>Hello {this.props.message}</div>;
  }
}

ReactDOM.render(
  <Greeting message="World" />,
  document.getElementById('root')
);
```

You can write the same code without JSX as below,

```
class Greeting extends React.Component {
  render() {
    return React.createElement('div', null, `Hello ${this.props.message}`);
  }
}

ReactDOM.render(
  React.createElement(Greeting, {message: 'World'}, null),
  document.getElementById('root')
);
```

[↑ Back to Top](#)

287. What is diffing algorithm?

React needs to use algorithms to find out how to efficiently update the UI to match the most recent tree. The diffing algorithms is generating the minimum number of operations to transform one tree into another. However, the algorithms have a complexity in the order of $O(n^3)$ where n is the number of elements in the tree.

In this case, for displaying 1000 elements would require in the order of one billion comparisons. This is far too expensive. Instead, React implements a heuristic $O(n)$ algorithm based on two assumptions:

- i. Two elements of different types will produce different trees.
- ii. The developer can hint at which child elements may be stable across different renders with a key prop.

[↑ Back to Top](#)

288. What are the rules covered by diffing algorithm?

When diffing two trees, React first compares the two root elements. The behavior is different depending on the types of the root elements. It covers the below rules during reconciliation algorithm,

- i. **Elements Of Different Types:** Whenever the root elements have different types, React will tear down the old tree and build the new tree from scratch. For example, elements to , or from to of different types lead a full rebuild.
- ii. **DOM Elements Of The Same Type:** When comparing two React DOM elements of the same type, React looks at the attributes of both, keeps the same underlying DOM node, and only updates the changed attributes. Lets take an example with same DOM elements except className attribute,

```
<div className="show" title="ReactJS" />
```

```
<div className="hide" title="ReactJS" />
```

- iii. **Component Elements Of The Same Type:** When a component updates, the instance stays the same, so that state is maintained across renders. React updates the props of the underlying component instance to match the new element, and calls `componentWillReceiveProps()` and `componentWillUpdate()` on the underlying instance. After that, the `render()` method is called and the diff algorithm recurses on the previous result and the new result.
- iv. **Recurse On Children:** when recursing on the children of a DOM node, React just iterates over both lists of children at the same time and generates a mutation whenever there's a difference. For example, when adding an element at the end of the children, converting between these two trees works well.

```
<ul>
  <li>first</li>
  <li>second</li>
</ul>

<ul>
  <li>first</li>
  <li>second</li>
  <li>third</li>
</ul>
```

- v. **Handling keys:** React supports a `key` attribute. When children have keys, React uses the key to match children in the original tree with children in the subsequent tree. For example, adding a key can make the tree conversion efficient,

```
<ul>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>

<ul>
  <li key="2014">Connecticut</li>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>
```

 [Back to Top](#)

289. When do you need to use refs?

There are few use cases to go for refs,

- i. Managing focus, text selection, or media playback.
- ii. Triggering imperative animations.

iii. Integrating with third-party DOM libraries.

 [Back to Top](#)

290. Is it prop must be named as render for render props?

Even though the pattern named render props, you don't have to use a prop named render to use this pattern. i.e, Any prop that is a function that a component uses to know what to render is technically a "render prop". Lets take an example with the children prop for render props,

```
<Mouse children={mouse => (
  <p>The mouse position is {mouse.x}, {mouse.y}</p>
)}>
```

Actually children prop doesn't need to be named in the list of "attributes" in JSX element. Instead, you can keep it directly inside element,

```
<Mouse>
  {mouse => (
    <p>The mouse position is {mouse.x}, {mouse.y}</p>
  )}
</Mouse>
```

While using this above technique(without any name), explicitly state that children should be a function in your propTypes.

```
Mouse.propTypes = {
  children: PropTypes.func.isRequired
};
```

 [Back to Top](#)

291. What are the problems of using render props with pure components?

If you create a function inside a render method, it negates the purpose of pure component. Because the shallow prop comparison will always return false for new props, and each render in this case will generate a new value for the render prop. You can solve this issue by defining the render function as instance method.

 [Back to Top](#)

292. How do you create HOC using render props?

You can implement most higher-order components (HOC) using a regular component with a render prop. For example, if you would prefer to have a withMouse HOC instead of a component, you could easily create one using a regular with a render prop.

```
function withMouse(Component) {
  return class extends React.Component {
    render() {
      return (
        <Mouse render={mouse => (
          <Component {...this.props} mouse={mouse} />
        )}/>
      );
    }
  }
}
```

This way render props gives the flexibility of using either pattern.

[↑ Back to Top](#)

293. What is windowing technique?

Windowing is a technique that only renders a small subset of your rows at any given time, and can dramatically reduce the time it takes to re-render the components as well as the number of DOM nodes created. If your application renders long lists of data then this technique is recommended. Both react-window and react-virtualized are popular windowing libraries which provides several reusable components for displaying lists, grids, and tabular data.

[↑ Back to Top](#)

294. How do you print falsy values in JSX?

The falsy values such as false, null, undefined, and true are valid children but they don't render anything. If you still want to display them then you need to convert it to string. Let's take an example on how to convert to a string,

```
<div>
  My JavaScript variable is {String(myVariable)}.
</div>
```

[↑ Back to Top](#)

295. What is the typical use case of portals?

React portals are very useful when a parent component has overflow: hidden or has properties that affect the stacking context(z-index,position,opacity etc styles) and you need to visually “break out” of its container.

For example, dialogs, global message notifications, hovercards, and tooltips.

 [Back to Top](#)

296. How do you set default value for uncontrolled component?

In React, the value attribute on form elements will override the value in the DOM. With an uncontrolled component, you might want React to specify the initial value, but leave subsequent updates uncontrolled. To handle this case, you can specify a **defaultValue** attribute instead of **value**.

```
render() {
  return (
    <form onSubmit={this.handleSubmit}>
      <label>
        User Name:
        <input
          defaultValue="John"
          type="text"
          ref={this.input} />
      </label>
      <input type="submit" value="Submit" />
    </form>
  );
}
```

The same applies for `select` and `textArea` inputs. But you need to use `defaultChecked` for `checkbox` and `radio` inputs.

 [Back to Top](#)

297. What is your favorite React stack?

Even though the tech stack varies from developer to developer, the most popular stack is used in react boilerplate project code. It mainly uses Redux and redux-saga for state management and asynchronous side-effects, react-router for routing purpose, styled-components for styling react components, axios for invoking REST api, and other supported stack such as webpack, reselect, ESNext, Babel. You can clone the project <https://github.com/react-boilerplate/react-boilerplate> and start working on any new react project.

 [Back to Top](#)

298. What is the difference between Real DOM and Virtual DOM?

Below are the main differences between Real DOM and Virtual DOM,

| Real DOM | Virtual DOM |
|--------------------------------------|--------------------------------------|
| Updates are slow | Updates are fast |
| DOM manipulation is very expensive. | DOM manipulation is very easy |
| You can update HTML directly. | You Can't directly update HTML |
| It causes too much of memory wastage | There is no memory wastage |
| Creates a new DOM if element updates | It updates the JSX if element update |

 [Back to Top](#)

299. How to add Bootstrap to a react application?

Bootstrap can be added to your React app in a three possible ways,

- i. Using the Bootstrap CDN: This is the easiest way to add bootstrap. Add both bootstrap CSS and JS resources in a head tag.
- ii. Bootstrap as Dependency: If you are using a build tool or a module bundler such as Webpack, then this is the preferred option for adding Bootstrap to your React application

```
npm install bootstrap
```

- iii. React Bootstrap Package: In this case, you can add Bootstrap to our React app by using a package that has rebuilt Bootstrap components to work particularly as React components. Below packages are popular in this category,

- a. react-bootstrap
- b.reactstrap

 [Back to Top](#)

300. Can you list down top websites or applications using react as front end framework?

Below are the top 10 websites using React as their front-end framework,

- i. Facebook
- ii. Uber
- iii. Instagram
- iv. WhatsApp
- v. Khan Academy
- vi. Airbnb

- vii. Dropbox
- viii. Flipboard
- ix. Netflix
- x. PayPal

[!\[\]\(d91678734ce1b2ed8f4fe838d4090f47_img.jpg\) Back to Top](#)

301. Is it recommended to use CSS In JS technique in React?

React does not have any opinion about how styles are defined but if you are a beginner then good starting point is to define your styles in a separate *.css file as usual and refer to them using className. This functionality is not part of React but came from third-party libraries. But If you want to try a different approach(CSS-In-JS) then styled-components library is a good option.

[!\[\]\(33d05cac83ff08ab0854f3659ab61567_img.jpg\) Back to Top](#)

302. Do I need to rewrite all my class components with hooks?

No. But you can try Hooks in a few components(or new components) without rewriting any existing code. Because there are no plans to remove classes in ReactJS.

[!\[\]\(6e7400f06f72e71f27fec942953668e2_img.jpg\) Back to Top](#)

303. How to fetch data with React Hooks?

The effect hook called `useEffect` is used to fetch the data with axios from the API and to set the data in the local state of the component with the state hook's update function.

Let's take an example in which it fetches list of react articles from the API

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

function App() {
  const [data, setData] = useState({ hits: [] });

  useEffect(async () => {
    const result = await axios(
      'http://hn.algolia.com/api/v1/search?query=react',
    );

    setData(result.data);
  }, []);

  return (
    <ul>
      {data.hits.map(item => (
        <li key={item.objectID}>
          {item.title}
        </li>
      ))}
    </ul>
  );
}

export default App;
```

```

        <li key={item.objectID}>
          <a href={item.url}>{item.title}</a>
        </li>
      )}
    </ul>
  );
}

export default App;

```

Remember we provided an empty array as second argument to the effect hook to avoid activating it on component updates but only for the mounting of the component. i.e, It fetches only for component mount.

 [Back to Top](#)

304. Is Hooks cover all use cases for classes?

Hooks doesn't cover all use cases of classes but there is a plan to add them soon. Currently there are no Hook equivalents to the uncommon `getSnapshotBeforeUpdate` and `componentDidCatch` lifecycles yet.

 [Back to Top](#)

305. What is the stable release for hooks support?

React includes a stable implementation of React Hooks in 16.8 release for below packages

- i. React DOM
- ii. React DOM Server
- iii. React Test Renderer
- iv. React Shallow Renderer

 [Back to Top](#)

306. Why do we use array destructuring (square brackets notation) in `useState` ?

When we declare a state variable with `useState` , it returns a pair — an array with two items. The first item is the current value, and the second is a function that updates the value. Using [0] and [1] to access them is a bit confusing because they have a specific meaning. This is why we use array destructuring instead.

For example, the array index access would look as follows:

```
var userStateVariable = useState('userProfile'); // Returns an array pair
```

```
var user = userStateVariable[0]; // Access first item  
var setUser = userStateVariable[1]; // Access second item
```

Whereas with array destructuring the variables can be accessed as follows:

```
const [user, setUser] = useState('userProfile');
```

 [Back to Top](#)

307. What are the sources used for introducing hooks?

Hooks got the ideas from several different sources. Below are some of them,

- i. Previous experiments with functional APIs in the react-future repository
- ii. Community experiments with render prop APIs such as Reactions Component
- iii. State variables and state cells in DisplayScript.
- iv. Subscriptions in Rx.
- v. Reducer components in ReasonReact.

 [Back to Top](#)

308. How do you access imperative API of web components?

Web Components often expose an imperative API to implement its functions. You will need to use a **ref** to interact with the DOM node directly if you want to access imperative API of a web component. But if you are using third-party Web Components, the best solution is to write a React component that behaves as a **wrapper** for your Web Component.

 [Back to Top](#)

309. What is formik?

Formik is a small react form library that helps you with the three major problems,

- i. Getting values in and out of form state
- ii. Validation and error messages
- iii. Handling form submission

 [Back to Top](#)

310. What are typical middleware choices for handling asynchronous calls in Redux?

Some of the popular middleware choices for handling asynchronous calls in Redux eco

system are Redux Thunk, Redux Promise, Redux Saga .

[↑ Back to Top](#)

311. Do browsers understand JSX code?

No, browsers can't understand JSX code. You need a transpiler to convert your JSX to regular Javascript that browsers can understand. The most widely used transpiler right now is Babel.

[↑ Back to Top](#)

312. Describe about data flow in react?

React implements one-way reactive data flow using props which reduce boilerplate and is easier to understand than traditional two-way data binding.

[↑ Back to Top](#)

313. What is react scripts?

The `react-scripts` package is a set of scripts from the `create-react-app` starter pack which helps you kick off projects without configuring. The `react-scripts start` command sets up the development environment and starts a server, as well as hot module reloading.

[↑ Back to Top](#)

314. What are the features of create react app?

Below are the list of some of the features provided by create react app.

- i. React, JSX, ES6, Typescript and Flow syntax support.
- ii. Autoprefixed CSS
- iii. CSS Reset/Normalize
- iv. A live development server
- v. A fast interactive unit test runner with built-in support for coverage reporting
- vi. A build script to bundle JS, CSS, and images for production, with hashes and sourcemaps
- vii. An offline-first service worker and a web app manifest, meeting all the Progressive Web App criteria.

[↑ Back to Top](#)

315. What is the purpose of `renderToNodeStream` method?

The `ReactDOMServer#renderToNodeStream` method is used to generate HTML on the server and send the markup down on the initial request for faster page loads. It also helps search engines to crawl your pages easily for SEO purposes. **Note:** Remember this method is not available in the browser but only server.

 [Back to Top](#)

316. What is MobX?

MobX is a simple, scalable and battle tested state management solution for applying functional reactive programming (FTRP). For reactJs application, you need to install below packages,

```
npm install mobx --save  
npm install mobx-react --save
```

 [Back to Top](#)

317. What are the differences between Redux and MobX?

Below are the main differences between Redux and MobX,

| Topic | Redux | MobX |
|---------------|---|--|
| Definition | It is a javascript library for managing the application state | It is a library for reactively managing the state of your applications |
| Programming | It is mainly written in ES6 | It is written in JavaScript(ES5) |
| Data Store | There is only one large store exist for data storage | There is more than one store for storage |
| Usage | Mainly used for large and complex applications | Used for simple applications |
| Performance | Need to be improved | Provides better performance |
| How it stores | Uses JS Object to store | Uses observable to store the data |

 [Back to Top](#)

318. Should I learn ES6 before learning ReactJS?

No, you don't have to learn es2015/es6 to learn react. But you may find many resources or React ecosystem uses ES6 extensively. Let's see some of the frequently used ES6 features,

- i. **Destructuring:** To get props and use them in a component

```

// in es 5
var someData = this.props.someData
var dispatch = this.props.dispatch

// in es6
const { someData, dispatch } = this.props

```

ii. Spread operator: Helps in passing props down into a component

```

// in es 5
<SomeComponent someData={this.props.someData} dispatch={this.props.dispatch} .

// in es6
<SomeComponent {...this.props} />

```

iii. Arrow functions: Makes compact syntax

```

// es 5
var users = usersList.map(function (user) {
  return <li>{user.name}</li>
})
// es 6
const users = usersList.map(user => <li>{user.name}</li>);

```

[↑ Back to Top](#)

319. What is Concurrent Rendering?

The Concurrent rendering makes React apps to be more responsive by rendering component trees without blocking the main UI thread. It allows React to interrupt a long-running render to handle a high-priority event. i.e, When you enabled concurrent Mode, React will keep an eye on other tasks that need to be done, and if there's something with a higher priority it will pause what it is currently rendering and let the other task finish first. You can enable this in two ways,

```

// 1. Part of an app by wrapping with ConcurrentMode
<React.unstable_ConcurrentMode>
  <Something />
</React.unstable_ConcurrentMode>

// 2. Whole app using createRoot
ReactDOM.unstable_createRoot(domNode).render(<App />);

```

[↑ Back to Top](#)

320. What is the difference between async mode and concurrent mode?

Both refers the same thing. Previously concurrent Mode being referred to as "Async Mode" by React team. The name has been changed to highlight React's ability to perform work on different priority levels. So it avoids the confusion from other approaches to Async Rendering.

 [Back to Top](#)

321. Can I use javascript urls in react16.9?

Yes, you can use javascript: URLs but it will log a warning in the console. Because URLs starting with javascript: are dangerous by including unsanitized output in a tag like `<a href>` and create a security hole.

```
const companyProfile = {
  website: "javascript: alert('Your website is hacked')",
};
// It will log a warning
<a href={companyProfile.website}>More details</a>
```

Remember that the future versions will throw an error for javascript URLs.

 [Back to Top](#)

322. What is the purpose of eslint plugin for hooks?

The ESLint plugin enforces rules of Hooks to avoid bugs. It assumes that any function starting with "use" and a capital letter right after it is a Hook. In particular, the rule enforces that,

- i. Calls to Hooks are either inside a PascalCase function (assumed to be a component) or another useSomething function (assumed to be a custom Hook).
- ii. Hooks are called in the same order on every render.

 [Back to Top](#)

323. What is the difference between Imperative and Declarative in React?

Imagine a simple UI component, such as a "Like" button. When you tap it, it turns blue if it was previously grey, and grey if it was previously blue.

The imperative way of doing this would be:

```
if( user.likes() ) {
  if( hasBlue() ) {
    removeBlue();
    addGrey();
  } else {
```

```
        removeGrey();
        addBlue();
    }
}
```

Basically, you have to check what is currently on the screen and handle all the changes necessary to redraw it with the current state, including undoing the changes from the previous state. You can imagine how complex this could be in a real-world scenario.

In contrast, the declarative approach would be:

```
if( this.state.liked ) {
    return <blueLike />;
} else {
    return <greyLike />;
}
```

Because the declarative approach separates concerns, this part of it only needs to handle how the UI should look in a specific state, and is therefore much simpler to understand.

 [Back to Top](#)

324. What are the benefits of using typescript with reactjs?

Below are some of the benefits of using typescript with Reactjs,

- i. It is possible to use latest JavaScript features
- ii. Use of interfaces for complex type definitions
- iii. IDEs such as VS Code was made for TypeScript
- iv. Avoid bugs with the ease of readability and Validation

 [Back to Top](#)

325. How do you make sure that user remains authenticated on page refresh while using Context API State Management?

When a user logs in and reload, to persist the state generally we add the load user action in the useEffect hooks in the main App.js. While using Redux, loadUser action can be easily accessed.

App.js

```
import {loadUser} from '../actions/auth';
store.dispatch(loadUser());
```

- But while using **Context API**, to access context in App.js, wrap the AuthState in index.js

Releases so that App.js can access the auth context. Now whenever the page reloads, no matter what route you are on, the user will be authenticated as **loadUser** action will be triggered on each re-render.

index.js

Packages

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
import AuthState from './context/auth/AuthState'
```

Contributors 48

ReactSSTeam
ReactSSTeam <React.StrictMode> <App />
+ 37 contributors <AuthState>
</React.StrictMode>,
document.getElementById('root')



Languages

● JavaScript 71.6% ● HTML 18.4% ● CSS 10.0%

```
const authContext = useContext(AuthContext);

const { loadUser } = authContext;

useEffect(() => {
  loadUser();
}, [])
```

loadUser

```
const loadUser = async () => {
  const token = sessionStorage.getItem('token');

  if(!token){
    dispatch({
      type: ERROR
    })
  }
  setAuthToken(token);

  try {
    const res = await axios('/api/auth');

    dispatch({
      type: USER_LOADED,
      payload: res.data.data
  })
```

```
    } catch (err) {
      console.error(err);
    }
}
```

 [Back to Top](#)

326. What are the benefits of new JSX transform?

There are three major benefits of new JSX transform,

- i. It is possible to use JSX without importing React packages
- ii. The compiled output might improve the bundle size in a small amount
- iii. The future improvements provides the flexibility to reduce the number of concepts to learn React.

327. How does new JSX transform different from old transform?

The new JSX transform doesn't require React to be in scope. i.e, You don't need to import React package for simple scenarios.

Let's take an example to look at the main differences between the old and the new transform,

Old Transform:

```
import React from 'react';

function App() {
  return <h1>Good morning!!</h1>;
}


```

Now JSX transform convert the above code into regular JavaScript as below,

```
import React from 'react';

function App() {
  return React.createElement('h1', null, 'Good morning!!');
}


```

New Transform:

The new JSX transform doesn't require any React imports

```
function App() {
```

```
    return <h1>Good morning!!</h1>;
}
```

Under the hood JSX transform compiles to below code

```
import {jsx as _jsx} from 'react/jsx-runtime';

function App() {
  return _jsx('h1', { children: 'Good morning!!' });
}
```

Note: You still need to import React to use Hooks.

328. How do you get redux scaffolding using create-react-app?

Redux team has provided official redux+js or redux+typescript templates for create-react-app project. The generated project setup includes,

- i. Redux Toolkit and React-Redux dependencies
- ii. Create and configure Redux store
- iii. React-Redux `<Provider>` passing the store to React components
- iv. Small "counter" example to demo how to add redux logic and React-Redux hooks API to interact with the store from components

The below commands need to be executed along with template option as below,

i. Javascript template:

```
npx create-react-app my-app --template redux
```

ii. Typescript template:

```
npx create-react-app my-app --template redux-typescript
```

329. What are React Server components?

React Server Component is a way to write React component that gets rendered in the server-side with the purpose of improving React app performance. These components allow us to load components from the backend.

Note: React Server Components is still under development and not recommended for production yet.

| No. | Questions |
|-----|--|
| 419 | How do you define instance and non-instance properties |
| 420 | What is the difference between isNaN and Number.isNaN? |
| 421 | How to invoke an IIFE without any extra brackets? |
| 422 | Is that possible to use expressions in switch cases? |
| 423 | What is the easiest way to ignore promise errors? |
| 424 | How do style the console output using CSS? |

1. What are the possible ways to create objects in JavaScript

There are many ways to create objects in javascript as below

i. Object constructor:

The simplest way to create an empty object is using the Object constructor. Currently this approach is not recommended.

```
var object = new Object();
```

ii. Object's create method:

The create method of Object creates a new object by passing the prototype object as a parameter

```
var object = Object.create(null);
```

iii. Object literal syntax:

The object literal syntax is equivalent to create method when it passes null as parameter

```
var object = {};
```

iv. Function constructor:

Create any function and apply the new operator to create object instances,

```
function Person(name){
  var object = {};
  object.name=name;
  object.age=21;
```

```
        return object;
    }
var object = new Person("Sudheer");
```

v. Function constructor with prototype:

This is similar to function constructor but it uses prototype for their properties and methods,

```
function Person(){}
Person.prototype.name = "Sudheer";
var object = new Person();
```

This is equivalent to an instance created with an object create method with a function prototype and then call that function with an instance and parameters as arguments.

```
function func {};
new func(x, y, z);
```

(OR)

```
// Create a new instance using function prototype.
var newInstance = Object.create(func.prototype)

// Call the function
var result = func.call(newInstance, x, y, z),

// If the result is a non-null object then use it otherwise just use the new :
console.log(result && typeof result === 'object' ? result : newInstance);
```

vi. ES6 Class syntax:

ES6 introduces class feature to create the objects

```
class Person {
    constructor(name) {
        this.name = name;
    }
}

var object = new Person("Sudheer");
```

vii. Singleton pattern:

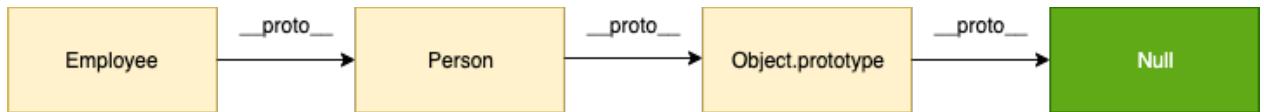
A Singleton is an object which can only be instantiated one time. Repeated calls to its constructor return the same instance and this way one can ensure that they don't accidentally create multiple instances.

```
var object = new function(){
    this.name = "Sudheer";
}
```

[↑ Back to Top](#)

2. What is a prototype chain

Prototype chaining is used to build new types of objects based on existing ones. It is similar to inheritance in a class based language. The prototype on object instance is available through `Object.getPrototypeOf(object)` or `proto` property whereas prototype on constructors function is available through `object.prototype`.



[↑ Back to Top](#)

3. What is the difference between Call, Apply and Bind

The difference between Call, Apply and Bind can be explained with below examples,

Call: The `call()` method invokes a function with a given `this` value and arguments provided one by one

```
var employee1 = {firstName: 'John', lastName: 'Rodson'};
var employee2 = {firstName: 'Jimmy', lastName: 'Baily'};

function invite(greeting1, greeting2) {
    console.log(greeting1 + ' ' + this.firstName + ' ' + this.lastName + ', ' + greet
}

invite.call(employee1, 'Hello', 'How are you?'); // Hello John Rodson, How are yo
invite.call(employee2, 'Hello', 'How are you?'); // Hello Jimmy Baily, How are yo
```

Apply: Invokes the function and allows you to pass in arguments as an array

```
var employee1 = {firstName: 'John', lastName: 'Rodson'};
var employee2 = {firstName: 'Jimmy', lastName: 'Baily'};

function invite(greeting1, greeting2) {
    console.log(greeting1 + ' ' + this.firstName + ' ' + this.lastName + ', ' + greet
}
```

```
invite.apply(employee1, ['Hello', 'How are you?']); // Hello John Rodson, How are  
invite.apply(employee2, ['Hello', 'How are you?']); // Hello Jimmy Baily, How are
```

bind: returns a new function, allowing you to pass in an array and any number of arguments

```
var employee1 = {firstName: 'John', lastName: 'Rodson'};  
var employee2 = {firstName: 'Jimmy', lastName: 'Baily'};  
  
function invite(greeting1, greeting2) {  
    console.log(greeting1 + ' ' + this.firstName + ' ' + this.lastName+ ', '+ gree  
}  
  
var inviteEmployee1 = invite.bind(employee1);  
var inviteEmployee2 = invite.bind(employee2);  
inviteEmployee1('Hello', 'How are you?'); // Hello John Rodson, How are you?  
inviteEmployee2('Hello', 'How are you?'); // Hello Jimmy Baily, How are you?
```

Call and apply are pretty interchangeable. Both execute the current function immediately. You need to decide whether it's easier to send in an array or a comma separated list of arguments. You can remember by treating Call is for comma (separated list) and Apply is for Array. Whereas Bind creates a new function that will have `this` set to the first parameter passed to bind().

 [Back to Top](#)

4. What is JSON and its common operations

JSON is a text-based data format following JavaScript object syntax, which was popularized by [Douglas Crockford](#). It is useful when you want to transmit data across a network and it is basically just a text file with an extension of `.json`, and a MIME type of `application/json` **Parsing:** Converting a string to a native object

```
JSON.parse(text)
```

Stringification: **converting a native object to a string so it can be transmitted across the network

```
JSON.stringify(object)
```

 [Back to Top](#)

5. What is the purpose of the array slice method

The **slice()** method returns the selected elements in an array as a new array object. It selects the elements starting at the given start argument, and ends at the given optional end argument without including the last element. If you omit the second argument then it selects till the end. Some of the examples of this method are,

```
let arrayIntegers = [1, 2, 3, 4, 5];
let arrayIntegers1 = arrayIntegers.slice(0,2); // returns [1,2]
let arrayIntegers2 = arrayIntegers.slice(2,3); // returns [3]
let arrayIntegers3 = arrayIntegers.slice(4); //returns [5]
```

Note: Slice method won't mutate the original array but it returns the subset as a new array.

[↑ Back to Top](#)

6. What is the purpose of the array splice method

The **splice()** method is used either adds/removes items to/from an array, and then returns the removed item. The first argument specifies the array position for insertion or deletion whereas the option second argument indicates the number of elements to be deleted. Each additional argument is added to the array. Some of the examples of this method are,

```
let arrayIntegersOriginal1 = [1, 2, 3, 4, 5];
let arrayIntegersOriginal2 = [1, 2, 3, 4, 5];
let arrayIntegersOriginal3 = [1, 2, 3, 4, 5];

let arrayIntegers1 = arrayIntegersOriginal1.splice(0,2); // returns [1, 2]; original
let arrayIntegers2 = arrayIntegersOriginal2.splice(3); // returns [4, 5]; original
let arrayIntegers3 = arrayIntegersOriginal3.splice(3, 1, "a", "b", "c"); //returns
```

Note: Splice method modifies the original array and returns the deleted array.

[↑ Back to Top](#)

7. What is the difference between slice and splice

Some of the major difference in a tabular form

| Slice | Splice |
|--|---------------------------------------|
| Doesn't modify the original array(immutable) | Modifies the original array(mutable) |
| Returns the subset of original array | Returns the deleted elements as array |

| Slice | Splice |
|--------------------------------------|---|
| Used to pick the elements from array | Used to insert or delete elements to/from array |

[↑ Back to Top](#)

8. How do you compare Object and Map

Objects are similar to **Maps** in that both let you set keys to values, retrieve those values, delete keys, and detect whether something is stored at a key. Due to this reason, Objects have been used as Maps historically. But there are important differences that make using a Map preferable in certain cases.

- i. The keys of an Object are Strings and Symbols, whereas they can be any value for a Map, including functions, objects, and any primitive.
- ii. The keys in Map are ordered while keys added to Object are not. Thus, when iterating over it, a Map object returns keys in order of insertion.
- iii. You can get the size of a Map easily with the size property, while the number of properties in an Object must be determined manually.
- iv. A Map is an iterable and can thus be directly iterated, whereas iterating over an Object requires obtaining its keys in some fashion and iterating over them.
- v. An Object has a prototype, so there are default keys in the map that could collide with your keys if you're not careful. As of ES5 this can be bypassed by using map = Object.create(null), but this is seldom done.
- vi. A Map may perform better in scenarios involving frequent addition and removal of key pairs.

[↑ Back to Top](#)

9. What is the difference between == and === operators

JavaScript provides both strict(==, !==) and type-converting(==, !=) equality comparison. The strict operators take type of variable in consideration, while non-strict operators make type correction/conversion based upon values of variables. The strict operators follow the below conditions for different types,

- i. Two strings are strictly equal when they have the same sequence of characters, same length, and same characters in corresponding positions.
- ii. Two numbers are strictly equal when they are numerically equal. i.e, Having the same number value. There are two special cases in this,
 - a. NaN is not equal to anything, including NaN.
 - b. Positive and negative zeros are equal to one another.
- iii. Two Boolean operands are strictly equal if both are true or both are false.

- iv. Two objects are strictly equal if they refer to the same Object.
- v. Null and Undefined types are not equal with ===, but equal with ==. i.e,
null==undefined --> false but null==undefined --> true

Some of the example which covers the above cases,

```
0 == false    // true
0 === false   // false
1 == "1"      // true
1 === "1"     // false
null == undefined // true
null === undefined // false
'0' == false // true
'0' === false // false
[]==[] or []===[ ] //false, refer different objects in memory
{}=={} or {}==={ } //false, refer different objects in memory
```

[↑ Back to Top](#)

10. What are lambda or arrow functions

An arrow function is a shorter syntax for a function expression and does not have its own **this, arguments, super, or new.target**. These functions are best suited for non-method functions, and they cannot be used as constructors.

[↑ Back to Top](#)

11. What is a first class function

In Javascript, functions are first class objects. First-class functions means when functions in that language are treated like any other variable.

For example, in such a language, a function can be passed as an argument to other functions, can be returned by another function and can be assigned as a value to a variable. For example, in the below example, handler functions assigned to a listener

```
const handler = () => console.log ('This is a click handler function');
document.addEventListener ('click', handler);
```

[↑ Back to Top](#)

12. What is a first order function

First-order function is a function that doesn't accept another function as an argument and doesn't return a function as its return value.

```
const firstOrder = () => console.log ('I am a first order function');
```

[↑ Back to Top](#)

13. What is a higher order function

Higher-order function is a function that accepts another function as an argument or returns a function as a return value.

```
const firstOrderFunc = () => console.log ('Hello I am a First order function');
const higherOrder = ReturnFirstOrderFunc => ReturnFirstOrderFunc ();
higherOrder (firstOrderFunc);
```

[↑ Back to Top](#)

14. What is a unary function

Unary function (i.e. monadic) is a function that accepts exactly one argument. Let us take an example of unary function. It stands for a single argument accepted by a function.

```
const unaryFunction = a => console.log (a + 10); // Add 10 to the given argument ↴
```

[↑ Back to Top](#)

15. What is the currying function

Currying is the process of taking a function with multiple arguments and turning it into a sequence of functions each with only a single argument. Currying is named after a mathematician Haskell Curry. By applying currying, a n-ary function turns it into a unary function. Let's take an example of n-ary function and how it turns into a currying function

```
const multiArgFunction = (a, b, c) => a + b + c;
const curryUnaryFunction = a => b => c => a + b + c;
curryUnaryFunction (1); // returns a function: b => c => 1 + b + c
curryUnaryFunction (1) (2); // returns a function: c => 3 + c
curryUnaryFunction (1) (2) (3); // returns the number 6
```

Curried functions are great to improve code reusability and functional composition.

[↑ Back to Top](#)

16. What is a pure function

A **Pure function** is a function where the return value is only determined by its arguments without any side effects. i.e, If you call a function with the same arguments 'n' number of times and 'n' number of places in the application then it will always return the same value. Let's take an example to see the difference between pure and impure functions,

```
//Impure
let numberArray = [];
const impureAddNumber = number => numberArray.push (number);
//Pure
const pureAddNumber = number => argNumberArray =>
    argNumberArray.concat ([number]);

//Display the results
console.log (impureAddNumber (6)); // returns 1
console.log (numberArray); // returns [6]
console.log (pureAddNumber (7) (numberArray)); // returns [6, 7]
console.log (numberArray); // returns [6]
```

As per above code snippets, Push function is impure itself by altering the array and returning an push number index which is independent of parameter value. Whereas Concat on the other hand takes the array and concatenates it with the other array producing a whole new array without side effects. Also, the return value is a concatenation of the previous array. Remember that Pure functions are important as they simplify unit testing without any side effects and no need for dependency injection. They also avoid tight coupling and make it harder to break your application by not having any side effects. These principles are coming together with **Immutability** concept of ES6 by giving preference to **const** over **let** usage.

[↑ Back to Top](#)

17. What is the purpose of the let keyword

The **let** statement declares a **block scope local variable**. Hence the variables defined with let keyword are limited in scope to the block, statement, or expression on which it is used. Whereas variables declared with the var keyword used to define a variable globally, or locally to an entire function regardless of block scope. Let's take an example to demonstrate the usage,

```
let counter = 30;
if (counter === 30) {
    let counter = 31;
    console.log(counter); // 31
}
console.log(counter); // 30 (because if block variable won't exist here)
```

[↑ Back to Top](#)

18. What is the difference between let and var

You can list out the differences in a tabular format

| var | let |
|---|-----------------------------|
| It is been available from the beginning of JavaScript | Introduced as part of ES6 |
| It has function scope | It has block scope |
| Variables will be hoisted | Hoisted but not initialized |

Let's take an example to see the difference,

```
function userDetails(username) {  
    if(username) {  
        console.log(salary); // undefined(due to hoisting)  
        console.log(age); // error: age is not defined  
        let age = 30;  
        var salary = 10000;  
    }  
    console.log(salary); //10000 (accessible to due function scope)  
    console.log(age); //error: age is not defined(due to block scope)  
}
```

 [Back to Top](#)

19. What is the reason to choose the name let as a keyword

Let is a mathematical statement that was adopted by early programming languages like Scheme and Basic. It has been borrowed from dozens of other languages that use let already as a traditional keyword as close to var as possible.

 [Back to Top](#)

20. How do you redeclare variables in switch block without an error

If you try to redeclare variables in a `switch` block then it will cause errors because there is only one block. For example, the below code block throws a syntax error as below,

```
let counter = 1;  
switch(x) {  
    case 0:  
        let name;  
        break;  
  
    case 1:  
        let name; // SyntaxError for redeclaration.  
        break;
```

```
}
```

To avoid this error, you can create a nested block inside a case clause and create a new block scoped lexical environment.

```
let counter = 1;
switch(x) {
  case 0: {
    let name;
    break;
  }
  case 1: {
    let name; // No SyntaxError for redeclaration.
    break;
  }
}
```

 [Back to Top](#)

21. What is the Temporal Dead Zone

The Temporal Dead Zone is a behavior in JavaScript that occurs when declaring a variable with the `let` and `const` keywords, but not with `var`. In ECMAScript 6, accessing a `let` or `const` variable before its declaration (within its scope) causes a `ReferenceError`. The time span when that happens, between the creation of a variable's binding and its declaration, is called the temporal dead zone. Let's see this behavior with an example,

```
function somemethod() {
  console.log(counter1); // undefined
  console.log(counter2); // ReferenceError
  var counter1 = 1;
  let counter2 = 2;
}
```

 [Back to Top](#)

22. What is IIFE(Immediately Invoked Function Expression)

IIFE (Immediately Invoked Function Expression) is a JavaScript function that runs as soon as it is defined. The signature of it would be as below,

```
(function ()  
{  
    // logic here  
}  
)  
();
```

The primary reason to use an IIFE is to obtain data privacy because any variables declared within the IIFE cannot be accessed by the outside world. i.e, If you try to access variables with IIFE then it throws an error as below,

```
(function ()  
{  
    var message = "IIFE";  
    console.log(message);  
}  
)  
();  
console.log(message); //Error: message is not defined
```

[↑ Back to Top](#)

23. What is the benefit of using modules

There are a lot of benefits to using modules in favour of a sprawling. Some of the benefits are,

- i. Maintainability
- ii. Reusability
- iii. Namespacing

[↑ Back to Top](#)

24. What is memoization

Memoization is a programming technique which attempts to increase a function's performance by caching its previously computed results. Each time a memoized function is called, its parameters are used to index the cache. If the data is present, then it can be returned, without executing the entire function. Otherwise the function is executed and then the result is added to the cache. Let's take an example of adding function with memoization,

```
const memoizAddition = () => {  
    let cache = {};  
    return (value) => {  
        if (value in cache) {
```

```

        console.log('Fetching from cache');
        return cache[value]; // Here, cache.value cannot be used as property name start
    }
    else {
        console.log('Calculating result');
        let result = value + 20;
        cache[value] = result;
        return result;
    }
}
}

// returned function from memoizAddition
const addition = memoizAddition();
console.log(addition(20)); //output: 40 calculated
console.log(addition(20)); //output: 40 cached

```

[↑ Back to Top](#)

25. What is Hoisting

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution. Remember that JavaScript only hoists declarations, not initialisation. Let's take a simple example of variable hoisting,

```

console.log(message); //output : undefined
var message = 'The variable Has been hoisted';

```

The above code looks like as below to the interpreter,

```

var message;
console.log(message);
message = 'The variable Has been hoisted';

```

[↑ Back to Top](#)

26. What are classes in ES6

In ES6, Javascript classes are primarily syntactic sugar over JavaScript's existing prototype-based inheritance. For example, the prototype based inheritance written in function expression as below,

```

function Bike(model,color) {
    this.model = model;
    this.color = color;
}

Bike.prototype.getDetails = function() {

```

```
        return this.model + ' bike has' + this.color + ' color';
    };
}
```

Whereas ES6 classes can be defined as an alternative

```
class Bike{
    constructor(color, model) {
        this.color= color;
        this.model= model;
    }

    getDetails() {
        return this.model + ' bike has' + this.color + ' color';
    }
}
```

[↑ Back to Top](#)

27. What are closures

A closure is the combination of a function and the lexical environment within which that function was declared. i.e, It is an inner function that has access to the outer or enclosing function's variables. The closure has three scope chains

- i. Own scope where variables defined between its curly brackets
- ii. Outer function's variables
- iii. Global variables Let's take an example of closure concept,

```
function Welcome(name){
    var greetingInfo = function(message){
        console.log(message+ ' '+name);
    }
    return greetingInfo();
}
var myFunction = Welcome('John');
myFunction('Welcome '); //Output: Welcome John
myFunction('Hello Mr.'); //output: Hello Mr.John
```

As per the above code, the inner function(greetingInfo) has access to the variables in the outer function scope(Welcome) even after the outer function has returned.

[↑ Back to Top](#)

28. What are modules

Modules refer to small units of independent, reusable code and also act as the foundation of many JavaScript design patterns. Most of the JavaScript modules export an object literal, a function, or a constructor

 [Back to Top](#)

29. Why do you need modules

Below are the list of benefits using modules in javascript ecosystem

- i. Maintainability
- ii. Reusability
- iii. Namespacing

 [Back to Top](#)

30. What is scope in javascript

Scope is the accessibility of variables, functions, and objects in some particular part of your code during runtime. In other words, scope determines the visibility of variables and other resources in areas of your code.

 [Back to Top](#)

31. What is a service worker

A Service worker is basically a script (JavaScript file) that runs in the background, separate from a web page and provides features that don't need a web page or user interaction. Some of the major features of service workers are Rich offline experiences (offline first web application development), periodic background syncs, push notifications, intercept and handle network requests and programmatically managing a cache of responses.

 [Back to Top](#)

32. How do you manipulate DOM using a service worker

Service worker can't access the DOM directly. But it can communicate with the pages it controls by responding to messages sent via the `postMessage` interface, and those pages can manipulate the DOM.

 [Back to Top](#)

33. How do you reuse information across service worker restarts

The problem with service worker is that it gets terminated when not in use, and restarted when it's next needed, so you cannot rely on global state within a service worker's `onfetch` and `onmessage` handlers. In this case, service workers will have access to IndexedDB API in order to persist and reuse across restarts.

 [Back to Top](#)

34. What is IndexedDB

IndexedDB is a low-level API for client-side storage of larger amounts of structured data, including files/blobs. This API uses indexes to enable high-performance searches of this data.

 [Back to Top](#)

35. What is web storage

Web storage is an API that provides a mechanism by which browsers can store key/value pairs locally within the user's browser, in a much more intuitive fashion than using cookies. The web storage provides two mechanisms for storing data on the client.

- i. **Local storage:** It stores data for current origin with no expiration date.
- ii. **Session storage:** It stores data for one session and the data is lost when the browser tab is closed.

 [Back to Top](#)

36. What is a post message

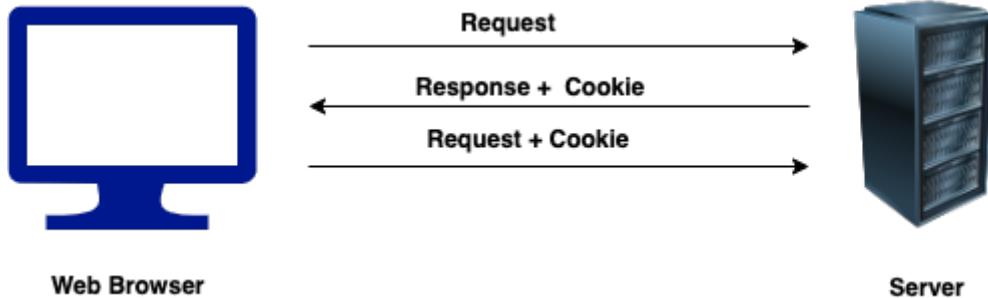
Post message is a method that enables cross-origin communication between Window objects.(i.e, between a page and a pop-up that it spawned, or between a page and an iframe embedded within it). Generally, scripts on different pages are allowed to access each other if and only if the pages follow same-origin policy(i.e, pages share the same protocol, port number, and host).

 [Back to Top](#)

37. What is a Cookie

A cookie is a piece of data that is stored on your computer to be accessed by your browser. Cookies are saved as key/value pairs. For example, you can create a cookie named `username` as below,

```
document.cookie = "username=John";
```



[↑ Back to Top](#)

38. Why do you need a Cookie

Cookies are used to remember information about the user profile(such as username). It basically involves two steps,

- i. When a user visits a web page, the user profile can be stored in a cookie.
- ii. Next time the user visits the page, the cookie remembers the user profile.

[↑ Back to Top](#)

39. What are the options in a cookie

There are few below options available for a cookie,

- i. By default, the cookie is deleted when the browser is closed but you can change this behavior by setting expiry date (in UTC time).

```
document.cookie = "username=John; expires=Sat, 8 Jun 2019 12:00:00 UTC";
```

- i. By default, the cookie belongs to a current page. But you can tell the browser what path the cookie belongs to using a path parameter.

```
document.cookie = "username=John; path=/services";
```

[↑ Back to Top](#)

40. How do you delete a cookie

You can delete a cookie by setting the expiry date as a passed date. You don't need to specify a cookie value in this case. For example, you can delete a username cookie in the current page as below.

```
document.cookie = "username=; expires=Fri, 07 Jun 2019 00:00:00 UTC; path=/;";
```

Note: You should define the cookie path option to ensure that you delete the right cookie. Some browsers doesn't allow to delete a cookie unless you specify a path parameter.

[↑ Back to Top](#)

41. What are the differences between cookie, local storage and session storage

Below are some of the differences between cookie, local storage and session storage,

| Feature | Cookie | Local storage | Session storage |
|-----------------------------------|------------------------------------|------------------|---------------------|
| Accessed on client or server side | Both server-side & client-side | client-side only | client-side only |
| Lifetime | As configured using Expires option | until deleted | until tab is closed |
| SSL support | Supported | Not supported | Not supported |
| Maximum data size | 4KB | 5 MB | 5MB |

[↑ Back to Top](#)

42. What is the main difference between localStorage and sessionStorage

LocalStorage is the same as SessionStorage but it persists the data even when the browser is closed and reopened(i.e it has no expiration time) whereas in sessionStorage data gets cleared when the page session ends.

[↑ Back to Top](#)

43. How do you access web storage

The Window object implements the `WindowLocalStorage` and `WindowSessionStorage` objects which has `localStorage` (`window.localStorage`) and `sessionStorage` (`window.sessionStorage`) properties respectively. These properties create an instance of the Storage object, through which data items can be set, retrieved and removed for a specific domain and storage type (session or local). For example, you can read and write on local storage objects as below

```
localStorage.setItem('logo', document.getElementById('logo').value);
localStorage.getItem('logo');
```

[↑ Back to Top](#)

44. What are the methods available on session storage

The session storage provided methods for reading, writing and clearing the session data

```
// Save data to sessionStorage
sessionStorage.setItem('key', 'value');

// Get saved data from sessionStorage
let data = sessionStorage.getItem('key');

// Remove saved data from sessionStorage
sessionStorage.removeItem('key');

// Remove all saved data from sessionStorage
sessionStorage.clear();
```

[↑ Back to Top](#)

45. What is a storage event and its event handler

The StorageEvent is an event that fires when a storage area has been changed in the context of another document. Whereas onstorage property is an EventHandler for processing storage events. The syntax would be as below

```
window.onstorage = functionRef;
```

Let's take the example usage of onstorage event handler which logs the storage key and it's values

```
window.onstorage = function(e) {
  console.log('The ' + e.key +
    ' key has been changed from ' + e.oldValue +
    ' to ' + e.newValue + '.');
};
```

[↑ Back to Top](#)

46. Why do you need web storage

Web storage is more secure, and large amounts of data can be stored locally, without affecting website performance. Also, the information is never transferred to the server. Hence this is a more recommended approach than Cookies.

[↑ Back to Top](#)

47. How do you check web storage browser support

You need to check browser support for localStorage and sessionStorage before using web storage,

```
if (typeof(Storage) !== "undefined") {  
    // Code for localStorage/sessionStorage.  
} else {  
    // Sorry! No Web Storage support..  
}
```

[↑ Back to Top](#)

48. How do you check web workers browser support

You need to check browser support for web workers before using it

```
if (typeof(Worker) !== "undefined") {  
    // code for Web worker support.  
} else {  
    // Sorry! No Web Worker support..  
}
```

[↑ Back to Top](#)

49. Give an example of a web worker

You need to follow below steps to start using web workers for counting example

- i. Create a Web Worker File: You need to write a script to increment the count value.
Let's name it as counter.js

```
let i = 0;  
  
function timedCount() {  
    i = i + 1;  
    postMessage(i);  
    setTimeout("timedCount()", 500);  
}  
  
timedCount();
```

Here postMessage() method is used to post a message back to the HTML page

- i. Create a Web Worker Object: You can create a web worker object by checking for browser support. Let's name this file as web_worker_example.js

```
if (typeof(w) == "undefined") {
```

```
w = new Worker("counter.js");
}
```

and we can receive messages from web worker

```
w.onmessage = function(event){
  document.getElementById("message").innerHTML = event.data;
};
```

- i. Terminate a Web Worker: Web workers will continue to listen for messages (even after the external script is finished) until it is terminated. You can use the terminate() method to terminate listening to the messages.

```
w.terminate();
```

- i. Reuse the Web Worker: If you set the worker variable to undefined you can reuse the code

```
w = undefined;
```

 [Back to Top](#)

50. What are the restrictions of web workers on DOM

WebWorkers don't have access to below javascript objects since they are defined in an external files

- i. Window object
- ii. Document object
- iii. Parent object

 [Back to Top](#)

51. What is a promise

A promise is an object that may produce a single value some time in the future with either a resolved value or a reason that it's not resolved(for example, network error). It will be in one of the 3 possible states: fulfilled, rejected, or pending.

The syntax of Promise creation looks like below,

```
const promise = new Promise(function(resolve, reject) {
  // promise description
})
```

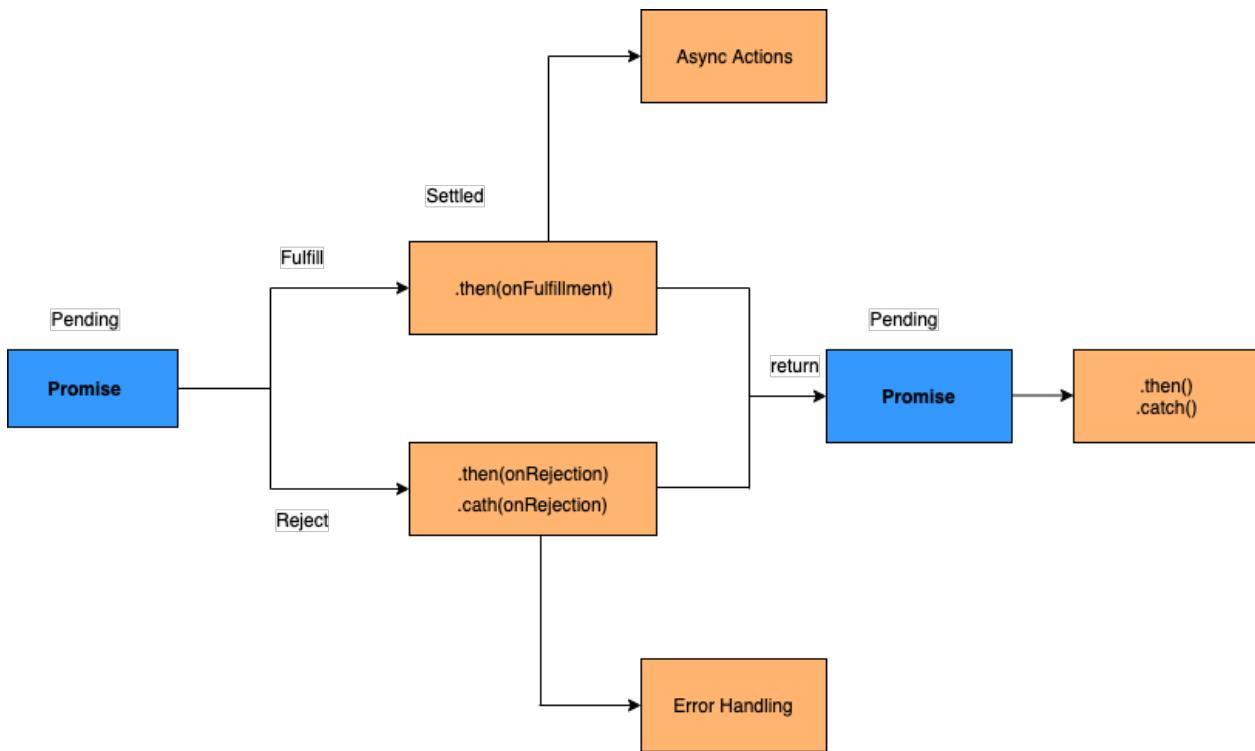
The usage of a promise would be as below,

```
const promise = new Promise(resolve => {
  setTimeout(() => {
    resolve("I'm a Promise!");
  }, 5000);
}, reject => {

});

promise.then(value => console.log(value));
```

The action flow of a promise will be as below,



[↑ Back to Top](#)

52. Why do you need a promise

Promises are used to handle asynchronous operations. They provide an alternative approach for callbacks by reducing the callback hell and writing the cleaner code.

[↑ Back to Top](#)

53. What are the three states of promise

Promises have three states:

- i. **Pending**: This is an initial state of the Promise before an operation begins
- ii. **Fulfilled**: This state indicates that the specified operation was completed.
- iii. **Rejected**: This state indicates that the operation did not complete. In this case an

error value will be thrown.

[↑ Back to Top](#)

54. What is a callback function

A callback function is a function passed into another function as an argument. This function is invoked inside the outer function to complete an action. Let's take a simple example of how to use callback function

```
function callbackFunction(name) {  
    console.log('Hello ' + name);  
}  
  
function outerFunction(callback) {  
    let name = prompt('Please enter your name.');//  
    callback(name);  
}  
  
outerFunction(callbackFunction);
```

[↑ Back to Top](#)

55. Why do we need callbacks

The callbacks are needed because javascript is an event driven language. That means instead of waiting for a response javascript will keep executing while listening for other events. Let's take an example with the first function invoking an API call(simulated by setTimeout) and the next function which logs the message.

```
function firstFunction(){  
    // Simulate a code delay  
    setTimeout( function(){  
        console.log('First function called');  
    }, 1000 );  
}  
  
function secondFunction(){  
    console.log('Second function called');  
}  
  
firstFunction();  
secondFunction();  
  
Output  
// Second function called  
// First function called
```

As observed from the output, javascript didn't wait for the response of the first function and the remaining code block got executed. So callbacks are used in a way to make sure that certain code doesn't execute until the other code finishes execution.

[↑ Back to Top](#)

56. What is a callback hell

Callback Hell is an anti-pattern with multiple nested callbacks which makes code hard to read and debug when dealing with asynchronous logic. The callback hell looks like below,

```
async1(function(){
    async2(function(){
        async3(function(){
            async4(function(){
                ....
            });
        });
    });
});
```

[↑ Back to Top](#)

57. What are server-sent events

Server-sent events (SSE) is a server push technology enabling a browser to receive automatic updates from a server via HTTP connection without resorting to polling. These are a one way communications channel - events flow from server to client only. This has been used in Facebook/Twitter updates, stock price updates, news feeds etc.

[↑ Back to Top](#)

58. How do you receive server-sent event notifications

The EventSource object is used to receive server-sent event notifications. For example, you can receive messages from server as below,

```
if(typeof(EventSource) !== "undefined") {
    var source = new EventSource("sse_generator.js");
    source.onmessage = function(event) {
        document.getElementById("output").innerHTML += event.data + "<br>";
    };
}
```

[↑ Back to Top](#)

59. How do you check browser support for server-sent events

You can perform browser support for server-sent events before using it as below,

```
if(typeof(EventSource) !== "undefined") {  
    // Server-sent events supported. Let's have some code here!  
} else {  
    // No server-sent events supported  
}
```

[↑ Back to Top](#)

60. What are the events available for server sent events

Below are the list of events available for server sent events

| Event | Description |
|-----------|--|
| onopen | It is used when a connection to the server is opened |
| onmessage | This event is used when a message is received |
| onerror | It happens when an error occurs |

[↑ Back to Top](#)

61. What are the main rules of promise

A promise must follow a specific set of rules,

- i. A promise is an object that supplies a standard-compliant `.then()` method
- ii. A pending promise may transition into either fulfilled or rejected state
- iii. A fulfilled or rejected promise is settled and it must not transition into any other state.
- iv. Once a promise is settled, the value must not change.

[↑ Back to Top](#)

62. What is callback in callback

You can nest one callback inside in another callback to execute the actions sequentially one by one. This is known as callbacks in callbacks.

```

loadScript('/script1.js', function(script) {
    console.log('first script is loaded');

    loadScript('/script2.js', function(script) {

        console.log('second script is loaded');

        loadScript('/script3.js', function(script) {

            console.log('third script is loaded');
            // after all scripts are loaded
        });
    });
});

```

[↑ Back to Top](#)

63. What is promise chaining

The process of executing a sequence of asynchronous tasks one after another using promises is known as Promise chaining. Let's take an example of promise chaining for calculating the final result,

```

new Promise(function(resolve, reject) {

    setTimeout(() => resolve(1), 1000);

}).then(function(result) {

    console.log(result); // 1
    return result * 2;

}).then(function(result) {

    console.log(result); // 2
    return result * 3;

}).then(function(result) {

    console.log(result); // 6
    return result * 4;

});

```

In the above handlers, the result is passed to the chain of .then() handlers with the below work flow,

- i. The initial promise resolves in 1 second,
- ii. After that `.then` handler is called by logging the result(1) and then return a promise with the value of result * 2.
- iii. After that the value passed to the next `.then` handler by logging the result(2) and return a promise with result * 3.
- iv. Finally the value passed to the last `.then` handler by logging the result(6) and return a promise with result * 4.

[↑ Back to Top](#)

64. What is promise.all

Promise.all is a promise that takes an array of promises as an input (an iterable), and it gets resolved when all the promises get resolved or any one of them gets rejected. For example, the syntax of promise.all method is below,

```
Promise.all([Promise1, Promise2, Promise3]) .then(result) => { console.log(resu]
```

Note: Remember that the order of the promises(output the result) is maintained as per input order.

[↑ Back to Top](#)

65. What is the purpose of the race method in promise

Promise.race() method will return the promise instance which is firstly resolved or rejected. Let's take an example of race() method where promise2 is resolved first

```
var promise1 = new Promise(function(resolve, reject) {
  setTimeout(resolve, 500, 'one');
});
var promise2 = new Promise(function(resolve, reject) {
  setTimeout(resolve, 100, 'two');
});

Promise.race([promise1, promise2]).then(function(value) {
  console.log(value); // "two" // Both promises will resolve, but promise2 is fast
});
```

[↑ Back to Top](#)

66. What is a strict mode in javascript

Strict Mode is a new feature in ECMAScript 5 that allows you to place a program, or a function, in a "strict" operating context. This way it prevents certain actions from being taken and throws more exceptions. The literal expression "use strict"; instructs the browser to use the javascript code in the Strict mode.

[↑ Back to Top](#)

67. Why do you need strict mode

Strict mode is useful to write "secure" JavaScript by notifying "bad syntax" into real errors. For example, it eliminates accidentally creating a global variable by throwing an error and also throws an error for assignment to a non-writable property, a getter-only property, a non-existing property, a non-existing variable, or a non-existing object.

[↑ Back to Top](#)

68. How do you declare strict mode

The strict mode is declared by adding "use strict"; to the beginning of a script or a function. If declared at the beginning of a script, it has global scope.

```
"use strict";
x = 3.14; // This will cause an error because x is not declared
```

and if you declare inside a function, it has local scope

```
x = 3.14;      // This will not cause an error.
myFunction();

function myFunction() {
  "use strict";
  y = 3.14;    // This will cause an error
}
```

[↑ Back to Top](#)

69. What is the purpose of double exclamation

The double exclamation or negation (!!) ensures the resulting type is a boolean. If it was falsey (e.g. 0, null, undefined, etc.), it will be false, otherwise, true. For example, you can test IE version using this expression as below,

```
let isIE8 = false;
isIE8 = !! navigator.userAgent.match(/MSIE 8.0/);
console.log(isIE8); // returns true or false
```

If you don't use this expression then it returns the original value.

```
console.log(navigator.userAgent.match(/MSIE 8.0/)); // returns either an Array or
```

Note: The expression !! is not an operator, but it is just twice of ! operator.

 [Back to Top](#)

70. What is the purpose of the delete operator

The delete keyword is used to delete the property as well as its value.

```
var user= {name: "John", age:20};  
delete user.age;  
  
console.log(user); // {name: "John"}
```

 [Back to Top](#)

71. What is the typeof operator

You can use the JavaScript typeof operator to find the type of a JavaScript variable. It returns the type of a variable or an expression.

```
typeof "John Abraham"      // Returns "string"  
typeof (1 + 2)            // Returns "number"
```

 [Back to Top](#)

72. What is undefined property

The undefined property indicates that a variable has not been assigned a value, or not declared at all. The type of undefined value is undefined too.

```
var user;    // Value is undefined, type is undefined  
console.log(typeof(user)) //undefined
```

Any variable can be emptied by setting the value to undefined.

```
user = undefined
```

 [Back to Top](#)

73. What is null value

The value null represents the intentional absence of any object value. It is one of JavaScript's primitive values. The type of null value is object. You can empty the variable by setting the value to null.

```
var user = null;  
console.log(typeof(user)) //object
```

[↑ Back to Top](#)

74. What is the difference between null and undefined

Below are the main differences between null and undefined,

| Null | Undefined |
|---|---|
| It is an assignment value which indicates that variable points to no object. | It is not an assignment value where a variable has been declared but has not yet been assigned a value. |
| Type of null is object | Type of undefined is undefined |
| The null value is a primitive value that represents the null, empty, or non-existent reference. | The undefined value is a primitive value used when a variable has not been assigned a value. |
| Indicates the absence of a value for a variable | Indicates absence of variable itself |
| Converted to zero (0) while performing primitive operations | Converted to NaN while performing primitive operations |

[↑ Back to Top](#)

75. What is eval

The eval() function evaluates JavaScript code represented as a string. The string can be a JavaScript expression, variable, statement, or sequence of statements.

```
console.log(eval('1 + 2')); // 3
```

[↑ Back to Top](#)

76. What is the difference between window and document

Below are the main differences between window and document,

| Window | Document |
|---|---|
| It is the root level element in any web page | It is the direct child of the window object. This is also known as Document Object Model(DOM) |
| By default window object is available implicitly in the page | You can access it via window.document or document. |
| It has methods like alert(), confirm() and properties like document, location | It provides methods like getElementById, getElementByTagName, createElement etc |

[!\[\]\(79d9c71a37c95a09a645dd2edb920a8a_img.jpg\) Back to Top](#)

77. How do you access history in javascript

The window.history object contains the browser's history. You can load previous and next URLs in the history using back() and next() methods.

```
function goBack() {
    window.history.back()
}
function goForward() {
    window.history.forward()
}
```

Note: You can also access history without window prefix.

[!\[\]\(9397ee8a33f64727fd0fdce8ebb09e7e_img.jpg\) Back to Top](#)

78. What are the javascript data types

Below are the list of javascript data types available

- i. Number
- ii. String
- iii. Boolean
- iv. Object
- v. Undefined

[!\[\]\(d2fe8796ff373382054c5960b8fc19c9_img.jpg\) Back to Top](#)

79. What is isNaN

The isNaN() function is used to determine whether a value is an illegal number (Not-a-Number) or not. i.e, This function returns true if the value equates to NaN. Otherwise it returns false.

```
isNaN('Hello') //true  
isNaN('100') //false
```

[↑ Back to Top](#)

80. What are the differences between undeclared and undefined variables

Below are the major differences between undeclared and undefined variables,

| undeclared | undefined |
|---|--|
| These variables do not exist in a program and are not declared | These variables declared in the program but have not assigned any value |
| If you try to read the value of an undeclared variable, then a runtime error is encountered | If you try to read the value of an undefined variable, an undefined value is returned. |

[↑ Back to Top](#)

81. What are global variables

Global variables are those that are available throughout the length of the code without any scope. The var keyword is used to declare a local variable but if you omit it then it will become global variable

```
msg = "Hello" // var is missing, it becomes global variable
```

[↑ Back to Top](#)

82. What are the problems with global variables

The problem with global variables is the conflict of variable names of local and global scope. It is also difficult to debug and test the code that relies on global variables.

[↑ Back to Top](#)

83. What is NaN property

The NaN property is a global property that represents "Not-a-Number" value. i.e, It indicates that a value is not a legal number. It is very rare to use NaN in a program but it can be used as return value for few cases

```
Math.sqrt(-1)  
parseInt("Hello")
```

 [Back to Top](#)

84. What is the purpose of isFinite function

The isFinite() function is used to determine whether a number is a finite, legal number. It returns false if the value is +infinity, -infinity, or NaN (Not-a-Number), otherwise it returns true.

```
isFinite(Infinity); // false  
isFinite(NaN); // false  
isFinite(-Infinity); // false  
  
isFinite(100); // true
```

 [Back to Top](#)

85. What is an event flow

Event flow is the order in which event is received on the web page. When you click an element that is nested in various other elements, before your click actually reaches its destination, or target element, it must trigger the click event for each of its parent elements first, starting at the top with the global window object. There are two ways of event flow

- i. Top to Bottom(Event Capturing)
- ii. Bottom to Top (Event Bubbling)

 [Back to Top](#)

86. What is event bubbling

Event bubbling is a type of event propagation where the event first triggers on the innermost target element, and then successively triggers on the ancestors (parents) of the target element in the same nesting hierarchy till it reaches the outermost DOM element.

 [Back to Top](#)

87. What is event capturing

Event capturing is a type of event propagation where the event is first captured by the outermost element, and then successively triggers on the descendants (children) of the target element in the same nesting hierarchy till it reaches the innermost DOM element.

 [Back to Top](#)

88. How do you submit a form using JavaScript

You can submit a form using JavaScript use `document.form[0].submit()`. All the form input's information is submitted using `onsubmit` event handler

```
function submit() {  
    document.form[0].submit();  
}
```

 [Back to Top](#)

89. How do you find operating system details

The `window.navigator` object contains information about the visitor's browser OS details. Some of the OS properties are available under `platform` property,

```
console.log(navigator.platform);
```

 [Back to Top](#)

90. What is the difference between document load and DOMContentLoaded events

The `DOMContentLoaded` event is fired when the initial HTML document has been completely loaded and parsed, without waiting for assets(stylesheets, images, and subframes) to finish loading. Whereas The `load` event is fired when the whole page has loaded, including all dependent resources(stylesheets, images).

 [Back to Top](#)

91. What is the difference between native, host and user objects

`Native objects` are objects that are part of the JavaScript language defined by the ECMAScript specification. For example, `String`, `Math`, `RegExp`, `Object`, `Function` etc core objects defined in the ECMAScript spec. `Host objects` are objects provided by the browser or runtime environment (Node). For example, `window`, `XmlHttpRequest`, `DOM nodes` etc are considered as host objects. `User objects` are objects defined in the javascript code. For example, User objects created for profile information.

 [Back to Top](#)

92. What are the tools or techniques used for debugging JavaScript code

You can use below tools or techniques for debugging javascript

- i. Chrome Devtools
- ii. debugger statement
- iii. Good old console.log statement

 [Back to Top](#)

93. What are the pros and cons of promises over callbacks

Below are the list of pros and cons of promises over callbacks,

Pros:

- i. It avoids callback hell which is unreadable
- ii. Easy to write sequential asynchronous code with .then()
- iii. Easy to write parallel asynchronous code with Promise.all()
- iv. Solves some of the common problems of callbacks(call the callback too late, too early, many times and swallow errors/exceptions)

Cons:

- i. It makes little complex code
- ii. You need to load a polyfill if ES6 is not supported

 [Back to Top](#)

94. What is the difference between an attribute and a property

Attributes are defined on the HTML markup whereas properties are defined on the DOM. For example, the below HTML element has 2 attributes type and value,

```
<input type="text" value="Name:">
```

You can retrieve the attribute value as below,

```
const input = document.querySelector('input');
console.log(input.getAttribute('value')); // Good morning
console.log(input.value); // Good morning
```

And after you change the value of the text field to "Good evening", it becomes like

```
console.log(input.getAttribute('value')); // Good morning  
console.log(input.value); // Good evening
```

 [Back to Top](#)

95. What is same-origin policy

The same-origin policy is a policy that prevents JavaScript from making requests across domain boundaries. An origin is defined as a combination of URI scheme, hostname, and port number. If you enable this policy then it prevents a malicious script on one page from obtaining access to sensitive data on another web page using Document Object Model(DOM).

 [Back to Top](#)

96. What is the purpose of void 0

Void(0) is used to prevent the page from refreshing. This will be helpful to eliminate the unwanted side-effect, because it will return the undefined primitive value. It is commonly used for HTML documents that use href="JavaScript:Void(0);" within an `<a>` element. i.e, when you click a link, the browser loads a new page or refreshes the same page. But this behavior will be prevented using this expression. For example, the below link notify the message without reloading the page

```
<a href="JavaScript:void(0);" onclick="alert('Well done!')">Click Me!</a>
```

 [Back to Top](#)

97. Is JavaScript a compiled or interpreted language

JavaScript is an interpreted language, not a compiled language. An interpreter in the browser reads over the JavaScript code, interprets each line, and runs it. Nowadays modern browsers use a technology known as Just-In-Time (JIT) compilation, which compiles JavaScript to executable bytecode just as it is about to run.

 [Back to Top](#)

98. Is JavaScript a case-sensitive language

Yes, JavaScript is a case sensitive language. The language keywords, variables, function & object names, and any other identifiers must always be typed with a consistent capitalization of letters.

 [Back to Top](#)

99. Is there any relation between Java and JavaScript

No, they are entirely two different programming languages and have nothing to do with each other. But both of them are Object Oriented Programming languages and like many other languages, they follow similar syntax for basic features(if, else, for, switch, break, continue etc).

 [Back to Top](#)

100. What are events

Events are "things" that happen to HTML elements. When JavaScript is used in HTML pages, JavaScript can react on these events. Some of the examples of HTML events are,

- i. Web page has finished loading
- ii. Input field was changed
- iii. Button was clicked

Let's describe the behavior of click event for button element,

```
<!doctype html>
<html>
  <head>
    <script>
      function greeting() {
        alert('Hello! Good morning');
      }
    </script>
  </head>
  <body>
    <button type="button" onclick="greeting()">Click me</button>
  </body>
</html>
```

 [Back to Top](#)

101. Who created javascript

JavaScript was created by Brendan Eich in 1995 during his time at Netscape Communications. Initially it was developed under the name Mocha , but later the language was officially called LiveScript when it first shipped in beta releases of Netscape.

 [Back to Top](#)

102. What is the use of preventDefault method

The preventDefault() method cancels the event if it is cancelable, meaning that the default action or behaviour that belongs to the event will not occur. For example, prevent form submission when clicking on submit button and prevent opening the page URL when clicking on hyperlink are some common use cases.

```
document.getElementById("link").addEventListener("click", function(event){  
    event.preventDefault();  
});
```

Note: Remember that not all events are cancelable.

 [Back to Top](#)

103. What is the use of stopPropagation method

The stopPropagation method is used to stop the event from bubbling up the event chain. For example, the below nested divs with stopPropagation method prevents default event propagation when clicking on nested div(Div1)

```
<p>Click DIV1 Element</p>  
<div onclick="secondFunc()">DIV 2  
    <div onclick="firstFunc(event)">DIV 1</div>  
</div>  
  
<script>  
function firstFunc(event) {  
    alert("DIV 1");  
    event.stopPropagation();  
}  
  
function secondFunc() {  
    alert("DIV 2");  
}  
</script>
```

 [Back to Top](#)

104. What are the steps involved in return false usage

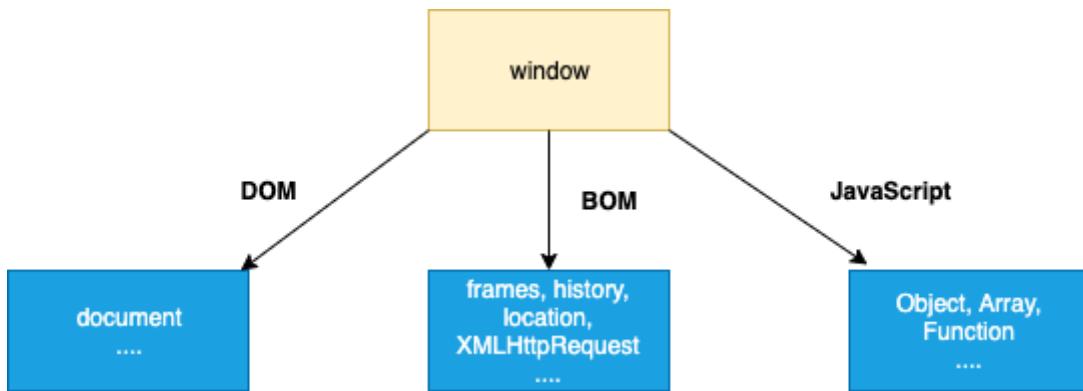
The return false statement in event handlers performs the below steps,

- i. First it stops the browser's default action or behaviour.
- ii. It prevents the event from propagating the DOM
- iii. Stops callback execution and returns immediately when called.

[↑ Back to Top](#)

105. What is BOM

The Browser Object Model (BOM) allows JavaScript to "talk to" the browser. It consists of the objects navigator, history, screen, location and document which are children of the window. The Browser Object Model is not standardized and can change based on different browsers.



[↑ Back to Top](#)

106. What is the use of setTimeout

The `setTimeout()` method is used to call a function or evaluate an expression after a specified number of milliseconds. For example, let's log a message after 2 seconds using `setTimeout` method,

```
setTimeout(function(){ console.log("Good morning"); }, 2000);
```

[↑ Back to Top](#)

107. What is the use of setInterval

The `setInterval()` method is used to call a function or evaluate an expression at specified intervals (in milliseconds). For example, let's log a message after 2 seconds using `setInterval` method,

```
setInterval(function(){ console.log("Good morning"); }, 2000);
```

[↑ Back to Top](#)

108. Why is JavaScript treated as Single threaded

JavaScript is a single-threaded language. Because the language specification does not allow the programmer to write code so that the interpreter can run parts of it in parallel in multiple threads or processes. Whereas languages like java, go, C++ can make multi-threaded and multi-process programs.

[↑ Back to Top](#)

109. What is an event delegation

Event delegation is a technique for listening to events where you delegate a parent element as the listener for all of the events that happen inside it.

For example, if you wanted to detect field changes in inside a specific form, you can use event delegation technique,

```
var form = document.querySelector('#registration-form');

// Listen for changes to fields inside the form
form.addEventListener('input', function (event) {

    // Log the field that was changed
    console.log(event.target);

}, false);
```

[↑ Back to Top](#)

110. What is ECMAScript

ECMAScript is the scripting language that forms the basis of JavaScript. ECMAScript standardized by the ECMA International standards organization in the ECMA-262 and ECMA-402 specifications. The first edition of ECMAScript was released in 1997.

[↑ Back to Top](#)

111. What is JSON

JSON (JavaScript Object Notation) is a lightweight format that is used for data interchanging. It is based on a subset of JavaScript language in the way objects are built in JavaScript.

[↑ Back to Top](#)

112. What are the syntax rules of JSON

Below are the list of syntax rules of JSON

- i. The data is in name/value pairs

- ii. The data is separated by commas
- iii. Curly braces hold objects
- iv. Square brackets hold arrays

 [Back to Top](#)

113. What is the purpose JSON stringify

When sending data to a web server, the data has to be in a string format. You can achieve this by converting JSON object into a string using `stringify()` method.

```
var userJSON = {'name': 'John', age: 31}  
var userString = JSON.stringify(user);  
console.log(userString); //>{"name":"John", "age":31}"
```

 [Back to Top](#)

114. How do you parse JSON string

When receiving the data from a web server, the data is always in a string format. But you can convert this string value to a javascript object using `parse()` method.

```
var userString = '{"name":"John", "age":31}';  
var userJSON = JSON.parse(userString);  
console.log(userJSON); // {name: "John", age: 31}
```

 [Back to Top](#)

115. Why do you need JSON

When exchanging data between a browser and a server, the data can only be text. Since JSON is text only, it can easily be sent to and from a server, and used as a data format by any programming language.

 [Back to Top](#)

116. What are PWAs

Progressive web applications (PWAs) are a type of mobile app delivered through the web, built using common web technologies including HTML, CSS and JavaScript. These PWAs are deployed to servers, accessible through URLs, and indexed by search engines.

 [Back to Top](#)

117. What is the purpose of clearTimeout method

The clearTimeout() function is used in javascript to clear the timeout which has been set by setTimeout() function before that. i.e, The return value of setTimeout() function is stored in a variable and it's passed into the clearTimeout() function to clear the timer.

For example, the below setTimeout method is used to display the message after 3 seconds. This timeout can be cleared by the clearTimeout() method.

```
<script>
var msg;
function greeting() {
    alert('Good morning');
}
function start() {
    msg =setTimeout(greeting, 3000);

}

function stop() {
    clearTimeout(msg);
}
</script>
```

 [Back to Top](#)

118. What is the purpose of clearInterval method

The clearInterval() function is used in javascript to clear the interval which has been set by setInterval() function. i.e, The return value returned by setInterval() function is stored in a variable and it's passed into the clearInterval() function to clear the interval.

For example, the below setInterval method is used to display the message for every 3 seconds. This interval can be cleared by the clearInterval() method.

```
<script>
var msg;
function greeting() {
    alert('Good morning');
}
function start() {
    msg = setInterval(greeting, 3000);

}

function stop() {
    clearInterval(msg);
}
</script>
```

[↑ Back to Top](#)

119. How do you redirect new page in javascript

In vanilla javascript, you can redirect to a new page using the `location` property of `window` object. The syntax would be as follows,

```
function redirect() {  
    window.location.href = 'newPage.html';  
}
```

[↑ Back to Top](#)

120. How do you check whether a string contains a substring

There are 3 possible ways to check whether a string contains a substring or not,

- i. **Using includes:** ES6 provided `String.prototype.includes` method to test a string contains a substring

```
var mainString = "hello", subString = "hell";  
mainString.includes(subString)
```

- i. **Using indexOf:** In an ES5 or older environment, you can use `String.prototype.indexOf` which returns the index of a substring. If the index value is not equal to -1 then it means the substring exists in the main string.

```
var mainString = "hello", subString = "hell";  
mainString.indexOf(subString) !== -1
```

- i. **Using RegEx:** The advanced solution is using Regular expression's test method(`RegExp.test`), which allows for testing for against regular expressions

```
var mainString = "hello", regex = /hell/;  
regex.test(mainString)
```

[↑ Back to Top](#)

121. How do you validate an email in javascript

You can validate an email in javascript using regular expressions. It is recommended to do validations on the server side instead of the client side. Because the javascript can be disabled on the client side.

```
function validateEmail(email) {  
    var re = /^(([^<>()\\[\\]\\.,;:\\s@"]+(\.[^<>()\\[\\]\\.,;:\\s@"]+)*|(.+))@((\\[[\r\n
```

```
        return re.test(String(email).toLowerCase());
    }
```

[↑ Back to Top](#)

The above regular expression accepts unicode characters.

122. How do you get the current url with javascript

You can use `window.location.href` expression to get the current url path and you can use the same expression for updating the URL too. You can also use `document.URL` for read-only purposes but this solution has issues in FF.

```
console.log('location.href', window.location.href); // Returns full URL
```

[↑ Back to Top](#)

123. What are the various url properties of location object

The below `Location` object properties can be used to access URL components of the page,

- i. `href` - The entire URL
- ii. `protocol` - The protocol of the URL
- iii. `host` - The hostname and port of the URL
- iv. `hostname` - The hostname of the URL
- v. `port` - The port number in the URL
- vi. `pathname` - The path name of the URL
- vii. `search` - The query portion of the URL
- viii. `hash` - The anchor portion of the URL

[↑ Back to Top](#)

124. How do get query string values in javascript

You can use `URLSearchParams` to get query string values in javascript. Let's see an example to get the client code value from URL query string,

```
const urlParams = new URLSearchParams(window.location.search);
const clientCode = urlParams.get('clientCode');
```

[↑ Back to Top](#)

125. How do you check if a key exists in an object

You can check whether a key exists in an object or not using three approaches,

- i. **Using in operator:** You can use the in operator whether a key exists in an object or not

```
"key" in obj
```

and If you want to check if a key doesn't exist, remember to use parenthesis,

```
!("key" in obj)
```

- i. **Using hasOwnProperty method:** You can use hasOwnProperty to particularly test for properties of the object instance (and not inherited properties)

```
obj.hasOwnProperty("key") // true
```

- i. **Using undefined comparison:** If you access a non-existing property from an object, the result is undefined. Let's compare the properties against undefined to determine the existence of the property.

```
const user = {  
    name: 'John'  
};  
  
console.log(user.name !== undefined);      // true  
console.log(user.nickName !== undefined); // false
```

 [Back to Top](#)

126. How do you loop through or enumerate javascript object

You can use the for-in loop to loop through javascript object. You can also make sure that the key you get is an actual property of an object, and doesn't come from the prototype using hasOwnProperty method.

```
var object = {  
    "k1": "value1",  
    "k2": "value2",  
    "k3": "value3"  
};  
  
for (var key in object) {  
    if (object.hasOwnProperty(key)) {  
        console.log(key + " -> " + object[key]); // k1 -> value1 ...  
    }  
}
```

 [Back to Top](#)

127. How do you test for an empty object

There are different solutions based on ECMAScript versions

- i. **Using Object.entries(ECMA 7+):** You can use object entries length along with constructor type.

```
Object.entries(obj).length === 0 && obj.constructor === Object // Since date object
```

- i. **Using Object.keys(ECMA 5+):** You can use object keys length along with constructor type.

```
Object.keys(obj).length === 0 && obj.constructor === Object // Since date object ]
```

- i. **Using for-in with hasOwnProperty(Pre-ECMA 5):** You can use a for-in loop along with hasOwnProperty.

```
function isEmpty(obj) {  
    for(var prop in obj) {  
        if(obj.hasOwnProperty(prop)) {  
            return false;  
        }  
    }  
  
    return JSON.stringify(obj) === JSON.stringify({});  
}
```

 [Back to Top](#)

128. What is an arguments object

The arguments object is an Array-like object accessible inside functions that contains the values of the arguments passed to that function. For example, let's see how to use arguments object inside sum function,

```
function sum() {  
    var total = 0;  
    for (var i = 0, len = arguments.length; i < len; ++i) {  
        total += arguments[i];  
    }  
    return total;  
}  
  
sum(1, 2, 3) // returns 6
```

Note: You can't apply array methods on arguments object. But you can convert into a regular array as below.

```
var argsArray = Array.prototype.slice.call(arguments);
```

[↑ Back to Top](#)

129. How do you make first letter of the string in an uppercase

You can create a function which uses a chain of string methods such as charAt, toUpperCase and slice methods to generate a string with the first letter in uppercase.

```
function capitalizeFirstLetter(string) {  
    return string.charAt(0).toUpperCase() + string.slice(1);  
}
```

[↑ Back to Top](#)

130. What are the pros and cons of for loop

The for-loop is a commonly used iteration syntax in javascript. It has both pros and cons
####Pros

- i. Works on every environment
- ii. You can use break and continue flow control statements

####Cons

- i. Too verbose
- ii. Imperative
- iii. You might face one-by-off errors

[↑ Back to Top](#)

131. How do you display the current date in javascript

You can use `new Date()` to generate a new Date object containing the current date and time. For example, let's display the current date in mm/dd/yyyy

```
var today = new Date();
var dd = String(today.getDate()).padStart(2, '0');
var mm = String(today.getMonth() + 1).padStart(2, '0'); //January is 0!
var yyyy = today.getFullYear();

today = mm + '/' + dd + '/' + yyyy;
document.write(today);
```

 [Back to Top](#)

132. How do you compare two date objects

You need to use date.getTime() method to compare date values instead of comparison operators (==, !=, ===, and !== operators)

```
var d1 = new Date();
var d2 = new Date(d1);
console.log(d1.getTime() === d2.getTime()); //True
console.log(d1 === d2); // False
```

 [Back to Top](#)

133. How do you check if a string starts with another string

You can use ECMAScript 6's `String.prototype.startsWith()` method to check if a string starts with another string or not. But it is not yet supported in all browsers. Let's see an example to see this usage,

```
"Good morning".startsWith("Good"); // true
"Good morning".startsWith("morning"); // false
```

 [Back to Top](#)

134. How do you trim a string in javascript

JavaScript provided a trim method on string types to trim any whitespaces present at the beginning or ending of the string.

```
"Hello World ".trim(); //Hello World
```

If your browser(<IE9) doesn't support this method then you can use below polyfill.

```
if (!String.prototype.trim) {
  (function() {
```

```
// Make sure we trim BOM and NBSP
var rtrim = /^[^\s\uFEFF\xA0]+|[\s\uFEFF\xA0]+$/g;
String.prototype.trim = function() {
    return this.replace(rtrim, '');
};
})();
}
```

[↑ Back to Top](#)

135. How do you add a key value pair in javascript

There are two possible solutions to add new properties to an object. Let's take a simple object to explain these solutions.

```
var object = {
    key1: value1,
    key2: value2
};
```

i. **Using dot notation:** This solution is useful when you know the name of the property

```
object.key3 = "value3";
```

i. **Using square bracket notation:** This solution is useful when the name of the property is dynamically determined.

```
obj["key3"] = "value3";
```

[↑ Back to Top](#)

136. Is the !-- notation represents a special operator

No, that's not a special operator. But it is a combination of 2 standard operators one after the other,

- i. A logical not (!)
- ii. A prefix decrement (--)

At first, the value decremented by one and then tested to see if it is equal to zero or not for determining the truthy/falsy value.

[↑ Back to Top](#)

137. How do you assign default values to variables

You can use the logical or operator `||` in an assignment expression to provide a default value. The syntax looks like as below,

```
var a = b || c;
```

As per the above expression, variable 'a' will get the value of 'c' only if 'b' is falsy (if is null, false, undefined, 0, empty string, or NaN), otherwise 'a' will get the value of 'b'.

[↑ Back to Top](#)

138. How do you define multiline strings

You can define multiline string literals using the " character followed by line terminator.

```
var str = "This is a \
very lengthy \
sentence!";
```

But if you have a space after the " character, the code will look exactly the same, but it will raise a SyntaxError.

[↑ Back to Top](#)

139. What is an app shell model

An application shell (or app shell) architecture is one way to build a Progressive Web App that reliably and instantly loads on your users' screens, similar to what you see in native applications. It is useful for getting some initial HTML to the screen fast without a network.

[↑ Back to Top](#)

140. Can we define properties for functions

Yes, We can define properties for functions because functions are also objects.

```
fn = function(x) {
    //Function code goes here
}

fn.name = "John";

fn.profile = function(y) {
    //Profile code goes here
}
```

[↑ Back to Top](#)

141. What is the way to find the number of parameters expected by a function

You can use `function.length` syntax to find the number of parameters expected by a function. Let's take an example of `sum` function to calculate the sum of numbers,

```
function sum(num1, num2, num3, num4){  
    return num1 + num2 + num3 + num4;  
}  
sum.length // 4 is the number of parameters expected.
```

[↑ Back to Top](#)

142. What is a polyfill

A polyfill is a piece of JS code used to provide modern functionality on older browsers that do not natively support it. For example, Silverlight plugin polyfill can be used to mimic the functionality of an HTML Canvas element on Microsoft Internet Explorer 7.

[↑ Back to Top](#)

143. What are break and continue statements

The break statement is used to "jump out" of a loop. i.e, It breaks the loop and continues executing the code after the loop.

```
for (i = 0; i < 10; i++) {  
    if (i === 5) { break; }  
    text += "Number: " + i + "<br>";  
}
```

The continue statement is used to "jump over" one iteration in the loop. i.e, It breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

```
for (i = 0; i < 10; i++) {  
    if (i === 5) { continue; }  
    text += "Number: " + i + "<br>";  
}
```

[↑ Back to Top](#)

144. What are js labels

The label statement allows us to name loops and blocks in JavaScript. We can then use these labels to refer back to the code later. For example, the below code with labels avoids printing the numbers when they are same,

```
var i, j;

loop1:
for (i = 0; i < 3; i++) {
    loop2:
    for (j = 0; j < 3; j++) {
        if (i === j) {
            continue loop1;
        }
        console.log('i = ' + i + ', j = ' + j);
    }
}

// Output is:
// "i = 1, j = 0"
// "i = 2, j = 0"
// "i = 2, j = 1"
```

 [Back to Top](#)

145. What are the benefits of keeping declarations at the top

It is recommended to keep all declarations at the top of each script or function. The benefits of doing this are,

- i. Gives cleaner code
- ii. It provides a single place to look for local variables
- iii. Easy to avoid unwanted global variables
- iv. It reduces the possibility of unwanted re-declarations

 [Back to Top](#)

146. What are the benefits of initializing variables

It is recommended to initialize variables because of the below benefits,

- i. It gives cleaner code
- ii. It provides a single place to initialize variables
- iii. Avoid undefined values in the code

 [Back to Top](#)

147. What are the recommendations to create new object

It is recommended to avoid creating new objects using `new Object()`. Instead you can initialize values based on its type to create the objects.

- i. Assign `{}` instead of `new Object()`
- ii. Assign `""` instead of `new String()`
- iii. Assign `0` instead of `new Number()`
- iv. Assign `false` instead of `new Boolean()`
- v. Assign `[]` instead of `new Array()`
- vi. Assign `/()/` instead of `new RegExp()`
- vii. Assign function `(){}` instead of `new Function()`

You can define them as an example,

```
var v1 = {};
var v2 = "";
var v3 = 0;
var v4 = false;
var v5 = [];
var v6 = /()/;
var v7 = function(){};
```

 [Back to Top](#)

148. How do you define JSON arrays

JSON arrays are written inside square brackets and arrays contain javascript objects. For example, the JSON array of users would be as below,

```
"users": [
    {"firstName":"John", "lastName":"Abrahm"},
    {"firstName":"Anna", "lastName":"Smith"},
    {"firstName":"Shane", "lastName":"Warn"}
]
```

 [Back to Top](#)

149. How do you generate random integers

You can use `Math.random()` with `Math.floor()` to return random integers. For example, if you want generate random integers between 1 to 10, the multiplication factor should be 10,

```
Math.floor(Math.random() * 10) + 1;      // returns a random integer from 1 to 10
Math.floor(Math.random() * 100) + 1;     // returns a random integer from 1 to 100
```

Note: Math.random() returns a random number between 0 (inclusive), and 1 (exclusive)

[↑ Back to Top](#)

150. Can you write a random integers function to print integers with in a range

Yes, you can create a proper random function to return a random number between min and max (both included)

```
function randomInteger(min, max) {  
    return Math.floor(Math.random() * (max - min + 1)) + min;  
}  
randomInteger(1, 100); // returns a random integer from 1 to 100  
randomInteger(1, 1000); // returns a random integer from 1 to 1000
```

[↑ Back to Top](#)

151. What is tree shaking

Tree shaking is a form of dead code elimination. It means that unused modules will not be included in the bundle during the build process and for that it relies on the static structure of ES2015 module syntax,(i.e. import and export). Initially this has been popularized by the ES2015 module bundler `rollup` .

[↑ Back to Top](#)

152. What is the need of tree shaking

Tree Shaking can significantly reduce the code size in any application. i.e, The less code we send over the wire the more performant the application will be. For example, if we just want to create a “Hello World” Application using SPA frameworks then it will take around a few MBs, but by tree shaking it can bring down the size to just a few hundred KBs. Tree shaking is implemented in Rollup and Webpack bundlers.

[↑ Back to Top](#)

153. Is it recommended to use eval

No, it allows arbitrary code to be run which causes a security problem. As we know that the eval() function is used to run text as code. In most of the cases, it should not be necessary to use it.

[↑ Back to Top](#)

154. What is a Regular Expression

A regular expression is a sequence of characters that forms a search pattern. You can use this search pattern for searching data in a text. These can be used to perform all types of text search and text replace operations. Let's see the syntax format now,

```
/pattern/modifiers;
```

For example, the regular expression or search pattern with case-insensitive username would be,

```
/John/i
```

 [Back to Top](#)

155. What are the string methods available in Regular expression

Regular Expressions has two string methods: search() and replace(). The search() method uses an expression to search for a match, and returns the position of the match.

```
var msg = "Hello John";
var n = msg.search(/John/i); // 6
```

The replace() method is used to return a modified string where the pattern is replaced.

```
var msg = "Hello John";
var n = msg.replace(/John/i, "Buttler"); // Hello Buttler
```

 [Back to Top](#)

156. What are modifiers in regular expression

Modifiers can be used to perform case-insensitive and global searches. Let's list down some of the modifiers,

| Modifier | Description |
|----------|---|
| i | Perform case-insensitive matching |
| g | Perform a global match rather than stops at first match |
| m | Perform multiline matching |

Let's take an example of global modifier,

```
var text = "Learn JS one by one";
```

```
var pattern = /one/g;
var result = text.match(pattern); // one,one
```

[↑ Back to Top](#)

157. What are regular expression patterns

Regular Expressions provide a group of patterns in order to match characters. Basically they are categorized into 3 types,

- i. **Brackets:** These are used to find a range of characters. For example, below are some use cases,
 - a. [abc]: Used to find any of the characters between the brackets(a,b,c)
 - b. [0-9]: Used to find any of the digits between the brackets
 - c. (a|b): Used to find any of the alternatives separated with |
- ii. **Metacharacters:** These are characters with a special meaning For example, below are some use cases,
 - a. \d: Used to find a digit
 - b. \s: Used to find a whitespace character
 - c. \b: Used to find a match at the beginning or ending of a word
- iii. **Quantifiers:** These are useful to define quantities For example, below are some use cases,
 - a. n+: Used to find matches for any string that contains at least one n
 - b. n*: Used to find matches for any string that contains zero or more occurrences of n
 - c. n?: Used to find matches for any string that contains zero or one occurrences of n

[↑ Back to Top](#)

158. What is a RegExp object

RegExp object is a regular expression object with predefined properties and methods. Let's see the simple usage of RegExp object,

```
var regexp = new RegExp('\\w+');
console.log(regexp);
// expected output: /\w+/"
```

[↑ Back to Top](#)

159. How do you search a string for a pattern

You can use the test() method of regular expression in order to search a string for a pattern, and return true or false depending on the result.

```
var pattern = /you/;  
console.log(pattern.test("How are you?")); //true
```

 [Back to Top](#)

160. What is the purpose of exec method

The purpose of exec method is similar to test method but it executes a search for a match in a specified string and returns a result array, or null instead of returning true/false.

```
var pattern = /you/;  
console.log(pattern.exec("How are you?")); //["you", index: 8, input: "How are yo
```

 [Back to Top](#)

161. How do you change the style of a HTML element

You can change inline style or classname of a HTML element using javascript

- i. **Using style property:** You can modify inline style using style property

```
document.getElementById("title").style.fontSize = "30px";
```

- i. **Using ClassName property:** It is easy to modify element class using className property

```
document.getElementById("title").style.className = "custom-title";
```

 [Back to Top](#)

162. What would be the result of 1+2+'3'

The output is going to be 33 . Since 1 and 2 are numeric values, the result of the first two digits is going to be a numeric value 3 . The next digit is a string type value because of that the addition of numeric value 3 and string type value 3 is just going to be a concatenation value 33 .

 [Back to Top](#)

163. What is a debugger statement

The debugger statement invokes any available debugging functionality, such as setting a breakpoint. If no debugging functionality is available, this statement has no effect. For example, in the below function a debugger statement has been inserted. So execution is paused at the debugger statement just like a breakpoint in the script source.

```
function getProfile() {  
    // code goes here  
    debugger;  
    // code goes here  
}
```

 [Back to Top](#)

164. What is the purpose of breakpoints in debugging

You can set breakpoints in the javascript code once the debugger statement is executed and the debugger window pops up. At each breakpoint, javascript will stop executing, and let you examine the JavaScript values. After examining values, you can resume the execution of code using the play button.

 [Back to Top](#)

165. Can I use reserved words as identifiers

No, you cannot use the reserved words as variables, labels, object or function names. Let's see one simple example,

```
var else = "hello"; // Uncaught SyntaxError: Unexpected token else
```

 [Back to Top](#)

166. How do you detect a mobile browser

You can use regex which returns a true or false value depending on whether or not the user is browsing with a mobile.

```
window.mobilecheck = function() {  
    var mobileCheck = false;  
    (function(a){if(/(android|bb\d+|meego).+mobile|avantgo|bada\/*|blackberry|blazer|  
        return mobileCheck;  
    };
```

 [Back to Top](#)

167. How do you detect a mobile browser without regexp

You can detect mobile browsers by simply running through a list of devices and checking if the useragent matches anything. This is an alternative solution for RegExp usage,

```
function detectmob() {
  if( navigator.userAgent.match(/Android/i)
  || navigator.userAgent.match(/webOS/i)
  || navigator.userAgent.match(/iPhone/i)
  || navigator.userAgent.match(/iPad/i)
  || navigator.userAgent.match(/iPod/i)
  || navigator.userAgent.match(/BlackBerry/i)
  || navigator.userAgent.match(/Windows Phone/i)
){
  return true;
}
else {
  return false;
}
}
```

 [Back to Top](#)

168. How do you get the image width and height using JS

You can programmatically get the image and check the dimensions(width and height) using Javascript.

```
var img = new Image();
img.onload = function() {
  console.log(this.width + 'x' + this.height);
}
img.src = 'http://www.google.com/intl/en_ALL/images/logo.gif';
```

 [Back to Top](#)

169. How do you make synchronous HTTP request

Browsers provide an XMLHttpRequest object which can be used to make synchronous HTTP requests from JavaScript

```
function httpGet(theUrl)
{
  var xmlhttpReq = new XMLHttpRequest();
  xmlhttpReq.open( "GET", theUrl, false ); // false for synchronous request
  xmlhttpReq.send( null );
  return xmlhttpReq.responseText;
}
```

[↑ Back to Top](#)

170. How do you make asynchronous HTTP request

Browsers provide an XMLHttpRequest object which can be used to make asynchronous HTTP requests from JavaScript by passing the 3rd parameter as true.

```
function httpGetAsync(theUrl, callback)
{
    var xmlhttpReq = new XMLHttpRequest();
    xmlhttpReq.onreadystatechange = function() {
        if (xmlhttpReq.readyState == 4 && xmlhttpReq.status == 200)
            callback(xmlhttpReq.responseText);
    }
    xmlhttp.open("GET", theUrl, true); // true for asynchronous
    xmlhttp.send(null);
}
```

[↑ Back to Top](#)

171. How do you convert date to another timezone in javascript

You can use the `toLocaleString()` method to convert dates in one timezone to another. For example, let's convert current date to British English timezone as below,

```
console.log(event.toLocaleString('en-GB', { timeZone: 'UTC' })); //29/06/2019, 09:
```

[↑ Back to Top](#)

172. What are the properties used to get size of window

You can use `innerWidth`, `innerHeight`, `clientWidth`, `clientHeight` properties of windows, document element and document body objects to find the size of a window. Let's use them combination of these properties to calculate the size of a window or document,

```
var width = window.innerWidth
|| document.documentElement.clientWidth
|| document.body.clientWidth;

var height = window.innerHeight
|| document.documentElement.clientHeight
|| document.body.clientHeight;
```

[↑ Back to Top](#)

173. What is a conditional operator in javascript

The conditional (ternary) operator is the only JavaScript operator that takes three operands which acts as a shortcut for if statements.

```
var isAuthenticated = false;
console.log(isAuthenticated ? 'Hello, welcome' : 'Sorry, you are not authenticated')
```

[Back to Top](#)

174. Can you apply chaining on conditional operator

Yes, you can apply chaining on conditional operators similar to if ... else if ... else if ... else chain. The syntax is going to be as below,

```
function traceValue(someParam) {
    return condition1 ? value1
        : condition2 ? value2
        : condition3 ? value3
        : value4;
}

// The above conditional operator is equivalent to:

function traceValue(someParam) {
    if (condition1) { return value1; }
    else if (condition2) { return value2; }
    else if (condition3) { return value3; }
    else { return value4; }
}
```

[Back to Top](#)

175. What are the ways to execute javascript after page load

You can execute javascript after page load in many different ways,

i. `window.onload`:

```
window.onload = function ...
```

i. `document.onload`:

```
document.onload = function ...
```

i. `body onload`:

```
<body onload="script();">
```

[↑ Back to Top](#)

176. What is the difference between proto and prototype

The `__proto__` object is the actual object that is used in the lookup chain to resolve methods, etc. Whereas `prototype` is the object that is used to build `__proto__` when you create an object with `new`

```
( new Employee ).__proto__ === Employee.prototype;  
( new Employee ).prototype === undefined;
```

[↑ Back to Top](#)

177. Give an example where do you really need semicolon

It is recommended to use semicolons after every statement in JavaScript. For example, in the below case it throws an error "... is not a function" at runtime due to missing semicolon.

```
// define a function  
var fn = function () {  
    //...  
} // semicolon missing at this line  
  
// then execute some code inside a closure  
(function () {  
    //...  
})();
```

and it will be interpreted as

```
var fn = function () {  
    //...  
}(function () {  
    //...  
})();
```

In this case, we are passing the second function as an argument to the first function and then trying to call the result of the first function call as a function. Hence, the second function will fail with a "... is not a function" error at runtime.

[↑ Back to Top](#)

178. What is a freeze method

The `freeze()` method is used to freeze an object. Freezing an object does not allow adding new properties to an object, prevents from removing and prevents changing the enumerability, configurability, or writability of existing properties. i.e, It returns the passed object and does not create a frozen copy.

```
const obj = {
  prop: 100
};

Object.freeze(obj);
obj.prop = 200; // Throws an error in strict mode

console.log(obj.prop); //100
```

Note: It causes a `TypeError` if the argument passed is not an object.

[↑ Back to Top](#)

179. What is the purpose of freeze method

Below are the main benefits of using freeze method,

- i. It is used for freezing objects and arrays.
- ii. It is used to make an object immutable.

[↑ Back to Top](#)

180. Why do I need to use freeze method

In the Object-oriented paradigm, an existing API contains certain elements that are not intended to be extended, modified, or re-used outside of their current context. Hence it works as the `final` keyword which is used in various languages.

[↑ Back to Top](#)

181. How do you detect a browser language preference

You can use `navigator` object to detect a browser language preference as below,

```
var language = navigator.languages && navigator.languages[0] || // Chrome / Firefo
  navigator.language || // All browsers
  navigator.userLanguage; // IE <= 10

console.log(language);
```

[↑ Back to Top](#)

182. How to convert string to title case with javascript

Title case means that the first letter of each word is capitalized. You can convert a string to title case using the below function,

```
function toTitleCase(str) {  
    return str.replace(  
        /\w\S*/g,  
        function(txt) {  
            return txt.charAt(0).toUpperCase() + txt.substr(1).toLowerCase();  
        }  
    );  
}  
toTitleCase("good morning john"); // Good Morning John
```

 [Back to Top](#)

183. How do you detect javascript disabled in the page

You can use the `<noscript>` tag to detect javascript disabled or not. The code block inside `<noscript>` gets executed when JavaScript is disabled, and is typically used to display alternative content when the page generated in JavaScript.

```
<script type="javascript">  
    // JS related code goes here  
</script>  
<noscript>  
    <a href="next_page.html?noJS=true">JavaScript is disabled in the page. Please  
</noscript>
```

 [Back to Top](#)

184. What are various operators supported by javascript

An operator is capable of manipulating(mathematical and logical computations) a certain value or operand. There are various operators supported by JavaScript as below,

- i. **Arithmetic Operators:** Includes + (Addition), - (Subtraction), * (Multiplication), / (Division), % (Modulus), ++ (Increment) and -- (Decrement)
- ii. **Comparison Operators:** Includes == (Equal), != (Not Equal), === (Equal with type), > (Greater than), >= (Greater than or Equal to), < (Less than), <= (Less than or Equal to)
- iii. **Logical Operators:** Includes && (Logical AND), || (Logical OR), !(Logical NOT)
- iv. **Assignment Operators:** Includes = (Assignment Operator), += (Add and Assignment Operator), -= (Subtract and Assignment Operator), *= (Multiply and Assignment), /= (Divide and Assignment), %= (Modules and Assignment)
- v. **Ternary Operators:** It includes conditional(: ?) Operator

vi. **typeof Operator:** It uses to find type of variable. The syntax looks like `typeof variable`

[↑ Back to Top](#)

185. What is a rest parameter

Rest parameter is an improved way to handle function parameters which allows us to represent an indefinite number of arguments as an array. The syntax would be as below,

```
function f(a, b, ...theArgs) {  
    // ...  
}
```

For example, let's take a sum example to calculate on dynamic number of parameters,

```
function total(...args){  
    let sum = 0;  
    for(let i of args){  
        sum+=i;  
    }  
    return sum;  
}  
  
console.log(fun(1,2)); //3  
console.log(fun(1,2,3)); //6  
console.log(fun(1,2,3,4)); //13  
console.log(fun(1,2,3,4,5)); //15
```

Note: Rest parameter is added in ES2015 or ES6

[↑ Back to Top](#)

186. What happens if you do not use rest parameter as a last argument

The rest parameter should be the last argument, as its job is to collect all the remaining arguments into an array. For example, if you define a function like below it doesn't make any sense and will throw an error.

```
function someFunc(a,...b,c){  
    //Your code goes here  
    return;  
}
```

[↑ Back to Top](#)

187. What are the bitwise operators available in javascript

Below are the list of bitwise logical operators used in JavaScript

- i. Bitwise AND (&)
- ii. Bitwise OR (|)
- iii. Bitwise XOR (^)
- iv. Bitwise NOT (~)
- v. Left Shift (<<)
- vi. Sign Propagating Right Shift (>>)
- vii. Zero fill Right Shift (>>>)

[↑ Back to Top](#)

188. What is a spread operator

Spread operator allows iterables(arrays / objects / strings) to be expanded into single arguments/elements. Let's take an example to see this behavior,

```
function calculateSum(x, y, z) {  
    return x + y + z;  
}  
  
const numbers = [1, 2, 3];  
  
console.log(calculateSum(...numbers)); // 6
```

[↑ Back to Top](#)

189. How do you determine whether object is frozen or not

`Object.isFrozen()` method is used to determine if an object is frozen or not. An object is frozen if all of the below conditions hold true,

- i. If it is not extensible.
- ii. If all of its properties are non-configurable.
- iii. If all its data properties are non-writable. The usage is going to be as follows,

```
const object = {  
    property: 'Welcome JS world'  
};  
Object.freeze(object);  
console.log(Object.isFrozen(object));
```

[↑ Back to Top](#)

190. How do you determine two values same or not using object

The `Object.is()` method determines whether two values are the same value. For example, the usage with different types of values would be,

```
Object.is('hello', 'hello');      // true
Object.is(window, window);      // true
Object.is([], []); // false
```

Two values are the same if one of the following holds:

- i. both undefined
- ii. both null
- iii. both true or both false
- iv. both strings of the same length with the same characters in the same order
- v. both the same object (means both object have same reference)
- vi. both numbers and both +0 both -0 both NaN both non-zero and both not NaN and both have the same value.

 [Back to Top](#)

191. What is the purpose of using object is method

Some of the applications of Object's `is` method are follows,

- i. It is used for comparison of two strings.
- ii. It is used for comparison of two numbers.
- iii. It is used for comparing the polarity of two numbers.
- iv. It is used for comparison of two objects.

 [Back to Top](#)

192. How do you copy properties from one object to other

You can use the `Object.assign()` method which is used to copy the values and properties from one or more source objects to a target object. It returns the target object which has properties and values copied from the target object. The syntax would be as below,

```
Object.assign(target, ...sources)
```

Let's take example with one source and one target object,

```
const target = { a: 1, b: 2 };
const source = { b: 3, c: 4 };

const returnedTarget = Object.assign(target, source);
```

```
console.log(target); // { a: 1, b: 3, c: 4 }

console.log(returnedTarget); // { a: 1, b: 3, c: 4 }
```

As observed in the above code, there is a common property(`b`) from source to target so it's value has been overwritten.

[↑ Back to Top](#)

193. What are the applications of assign method

Below are the some of main applications of `Object.assign()` method,

- i. It is used for cloning an object.
- ii. It is used to merge objects with the same properties.

[↑ Back to Top](#)

194. What is a proxy object

The Proxy object is used to define custom behavior for fundamental operations such as property lookup, assignment, enumeration, function invocation, etc. The syntax would be as follows,

```
var p = new Proxy(target, handler);
```

Let's take an example of proxy object,

```
var handler = {
  get: function(obj, prop) {
    return prop in obj ?
      obj[prop] :
      100;
  }
};

var p = new Proxy({}, handler);
p.a = 10;
p.b = null;

console.log(p.a, p.b); // 10, null
console.log('c' in p, p.c); // false, 100
```

In the above code, it uses `get` handler which define the behavior of the proxy when an operation is performed on it

[↑ Back to Top](#)

195. What is the purpose of seal method

The **Object.seal()** method is used to seal an object, by preventing new properties from being added to it and marking all existing properties as non-configurable. But values of present properties can still be changed as long as they are writable. Let's see the below example to understand more about seal() method

```
const object = {
    property: 'Welcome JS world'
};
Object.seal(object);
object.property = 'Welcome to object world';
console.log(Object.isSealed(object)); // true
delete object.property; // You cannot delete when sealed
console.log(object.property); //Welcome to object world
```

[↑ Back to Top](#)

196. What are the applications of seal method

Below are the main applications of Object.seal() method,

- i. It is used for sealing objects and arrays.
- ii. It is used to make an object immutable.

[↑ Back to Top](#)

197. What are the differences between freeze and seal methods

If an object is frozen using the **Object.freeze()** method then its properties become immutable and no changes can be made in them whereas if an object is sealed using the **Object.seal()** method then the changes can be made in the existing properties of the object.

[↑ Back to Top](#)

198. How do you determine if an object is sealed or not

The **Object.isSealed()** method is used to determine if an object is sealed or not. An object is sealed if all of the below conditions hold true

- i. If it is not extensible.
- ii. If all of its properties are non-configurable.
- iii. If it is not removable (but not necessarily non-writable). Let's see it in the action

```
const object = {
  property: 'Hello, Good morning'
};

Object.seal(object); // Using seal() method to seal the object

console.log(Object.isSealed(object));           // checking whether the object is sealed
```

[↑ Back to Top](#)

199. How do you get enumerable key and value pairs

The Object.entries() method is used to return an array of a given object's own enumerable string-keyed property [key, value] pairs, in the same order as that provided by a for...in loop. Let's see the functionality of object.entries() method in an example,

```
const object = {
  a: 'Good morning',
  b: 100
};

for (let [key, value] of Object.entries(object)) {
  console.log(` ${key}: ${value}`);
  // a: 'Good morning'
  // b: 100
}
```

Note: The order is not guaranteed as object defined.

[↑ Back to Top](#)

200. What is the main difference between Object.values and Object.entries method

The Object.values() method's behavior is similar to Object.entries() method but it returns an array of values instead [key,value] pairs.

```
const object = {
  a: 'Good morning',
  b: 100
};

for (let value of Object.values(object)) {
  console.log(` ${value}`);
  // 'Good morning'
  // 100
}
```

[↑ Back to Top](#)

201. How can you get the list of keys of any object

You can use the `Object.keys()` method which is used to return an array of a given object's own property names, in the same order as we get with a normal loop. For example, you can get the keys of a user object,

```
const user = {
  name: 'John',
  gender: 'male',
  age: 40
};

console.log(Object.keys(user)); //['name', 'gender', 'age']
```

[↑ Back to Top](#)

202. How do you create an object with prototype

The `Object.create()` method is used to create a new object with the specified prototype object and properties. i.e, It uses an existing object as the prototype of the newly created object. It returns a new object with the specified prototype object and properties.

```
const user = {
  name: 'John',
  printInfo: function () {
    console.log(`My name is ${this.name}`);
  }
};

const admin = Object.create(user);

admin.name = "Nick"; // Remember that "name" is a property set on "admin" but not
                     // on its prototype "user"

admin.printInfo(); // My name is Nick
```

[↑ Back to Top](#)

203. What is a WeakSet

`WeakSet` is used to store a collection of weakly(weak references) held objects. The syntax would be as follows,

```
new WeakSet([iterable]);
```

Let's see the below example to explain it's behavior,

```
var ws = new WeakSet();
var user = {};
ws.add(user);
ws.has(user);    // true
ws.delete(user); // removes user from the set
ws.has(user);    // false, user has been removed
```

[↑ Back to Top](#)

204. What are the differences between WeakSet and Set

The main difference is that references to objects in Set are strong while references to objects in WeakSet are weak. i.e, An object in WeakSet can be garbage collected if there is no other reference to it. Other differences are,

- i. Sets can store any value Whereas WeakSets can store only collections of objects
- ii. WeakSet does not have size property unlike Set
- iii. WeakSet does not have methods such as clear, keys, values, entries, forEach.
- iv. WeakSet is not iterable.

[↑ Back to Top](#)

205. List down the collection of methods available on WeakSet

Below are the list of methods available on WeakSet,

- i. add(value): A new object is appended with the given value to the weakset
- ii. delete(value): Deletes the value from the WeakSet collection.
- iii. has(value): It returns true if the value is present in the WeakSet Collection, otherwise it returns false.
- iv. length(): It returns the length of weakSetObject Let's see the functionality of all the above methods in an example,

```
var weakSetObject = new WeakSet();
var firstObject = {};
var secondObject = {};
// add(value)
weakSetObject.add(firstObject);
weakSetObject.add(secondObject);
console.log(weakSetObject.has(firstObject)); //true
console.log(weakSetObject.length()); //2
weakSetObject.delete(secondObject);
```

[↑ Back to Top](#)

206. What is a WeakMap

The WeakMap object is a collection of key/value pairs in which the keys are weakly referenced. In this case, keys must be objects and the values can be arbitrary values. The syntax is looking like as below,

```
new WeakMap([iterable])
```

Let's see the below example to explain it's behavior,

```
var ws = new WeakMap();
var user = {};
ws.set(user);
ws.has(user);    // true
ws.delete(user); // removes user from the map
ws.has(user);    // false, user has been removed
```

 [Back to Top](#)

207. What are the differences between WeakMap and Map

The main difference is that references to key objects in Map are strong while references to key objects in WeakMap are weak. i.e, A key object in WeakMap can be garbage collected if there is no other reference to it. Other differences are,

- i. Maps can store any key type Whereas WeakMaps can store only collections of key objects
- ii. WeakMap does not have size property unlike Map
- iii. WeakMap does not have methods such as clear, keys, values, entries, forEach.
- iv. WeakMap is not iterable.

 [Back to Top](#)

208. List down the collection of methods available on WeakMap

Below are the list of methods available on WeakMap,

- i. set(key, value): Sets the value for the key in the WeakMap object. Returns the WeakMap object.
- ii. delete(key): Removes any value associated to the key.
- iii. has(key): Returns a Boolean asserting whether a value has been associated to the key in the WeakMap object or not.
- iv. get(key): Returns the value associated to the key, or undefined if there is none. Let's see the functionality of all the above methods in an example,

```
var weakMapObject = new WeakMap();
var firstObject = {};
```

```
var secondObject = {};
// set(key, value)
weakMapObject.set(firstObject, 'John');
weakMapObject.set(secondObject, 100);
console.log(weakMapObject.has(firstObject)); //true
console.log(weakMapObject.get(firstObject)); // John
weakMapObject.delete(secondObject);
```

[↑ Back to Top](#)

209. What is the purpose of uneval

The `uneval()` is an inbuilt function which is used to create a string representation of the source code of an Object. It is a top-level function and is not associated with any object. Let's see the below example to know more about it's functionality,

```
var a = 1;
uneval(a); // returns a String containing 1
uneval(function user() {}); // returns "(function user(){})"
```

[↑ Back to Top](#)

210. How do you encode an URL

The `encodeURI()` function is used to encode complete URI which has special characters except (, / ? : @ & = + \$ #) characters.

```
var uri = 'https://mozilla.org/?x=шеллы';
var encoded = encodeURI(uri);
console.log(encoded); // https://mozilla.org/?x=%D1%88%D0%B5%D0%BB%D0%BB%D1%8B
```

[↑ Back to Top](#)

211. How do you decode an URL

The `decodeURI()` function is used to decode a Uniform Resource Identifier (URI) previously created by `encodeURI()`.

```
var uri = 'https://mozilla.org/?x=шеллы';
var encoded = encodeURI(uri);
console.log(encoded); // https://mozilla.org/?x=%D1%88%D0%B5%D0%BB%D0%BB%D1%8B
try {
  console.log(decodeURI(encoded)); // "https://mozilla.org/?x=шеллы"
} catch(e) { // catches a malformed URI
  console.error(e);
}
```

[↑ Back to Top](#)

212. How do you print the contents of web page

The window object provided a `print()` method which is used to print the contents of the current window. It opens a Print dialog box which lets you choose between various printing options. Let's see the usage of `print` method in an example,

```
<input type="button" value="Print" onclick="window.print()" />
```

Note: In most browsers, it will block while the print dialog is open.

[↑ Back to Top](#)

213. What is the difference between `uneval` and `eval`

The `uneval` function returns the source of a given object; whereas the `eval` function does the opposite, by evaluating that source code in a different memory area. Let's see an example to clarify the difference,

```
var msg = uneval(function greeting() { return 'Hello, Good morning'; });
var greeting = eval(msg);
greeting(); // returns "Hello, Good morning"
```

[↑ Back to Top](#)

214. What is an anonymous function

An anonymous function is a function without a name! Anonymous functions are commonly assigned to a variable name or used as a callback function. The syntax would be as below,

```
function (optionalParameters) {
    //do something
}

const myFunction = function(){ //Anonymous function assigned to a variable
    //do something
};

[1, 2, 3].map(function(element){ //Anonymous function used as a callback function
    //do something
});
```

Let's see the above anonymous function in an example,

```
var x = function (a, b) {return a * b};  
var z = x(5, 10);  
console.log(z); // 50
```

[↑ Back to Top](#)

215. What is the precedence order between local and global variables

A local variable takes precedence over a global variable with the same name. Let's see this behavior in an example.

```
var msg = "Good morning";  
function greeting() {  
    msg = "Good Evening";  
    console.log(msg);  
}  
greeting();
```

[↑ Back to Top](#)

216. What are javascript accessors

ECMAScript 5 introduced javascript object accessors or computed properties through getters and setters. Getters uses the `get` keyword whereas Setters uses the `set` keyword.

```
var user = {  
    firstName: "John",  
    lastName : "Abraham",  
    language : "en",  
    get lang() {  
        return this.language;  
    }  
    set lang(lang) {  
        this.language = lang;  
    }  
};  
console.log(user.lang); // getter access lang as en  
user.lang = 'fr';  
console.log(user.lang); // setter used to set lang as fr
```

[↑ Back to Top](#)

217. How do you define property on Object constructor

The `Object.defineProperty()` static method is used to define a new property directly on an object, or modify an existing property on an object, and returns the object. Let's see an example to know how to define property,

```
const newObject = {};  
  
Object.defineProperty(newObject, 'newProperty', {  
    value: 100,  
    writable: false  
});  
  
console.log(newObject.newProperty); // 100  
  
newObject.newProperty = 200; // It throws an error in strict mode due to writable
```

[↑ Back to Top](#)

218. What is the difference between get and defineProperty

Both have similar results until unless you use classes. If you use `get` the property will be defined on the prototype of the object whereas using `Object.defineProperty()` the property will be defined on the instance it is applied to.

[↑ Back to Top](#)

219. What are the advantages of Getters and Setters

Below are the list of benefits of Getters and Setters,

- i. They provide simpler syntax
- ii. They are used for defining computed properties, or accessors in JS.
- iii. Useful to provide equivalence relation between properties and methods
- iv. They can provide better data quality
- v. Useful for doing things behind the scenes with the encapsulated logic.

[↑ Back to Top](#)

220. Can I add getters and setters using defineProperty method

Yes, You can use the `Object.defineProperty()` method to add Getters and Setters. For example, the below counter object uses increment, decrement, add and subtract properties,

```
var obj = {counter : 0};  
  
// Define getters  
Object.defineProperty(obj, "increment", {
```

```

        get : function () {this.counter++;}
    });
Object.defineProperty(obj, "decrement", {
    get : function () {this.counter--;}
});

// Define setters
Object.defineProperty(obj, "add", {
    set : function (value) {this.counter += value;}
});
Object.defineProperty(obj, "subtract", {
    set : function (value) {this.counter -= value;}
});

obj.add = 10;
obj.subtract = 5;
console.log(obj.increment); //6
console.log(obj.decrement); //5

```

[↑ Back to Top](#)

221. What is the purpose of switch-case

The switch case statement in JavaScript is used for decision making purposes. In a few cases, using the switch case statement is going to be more convenient than if-else statements. The syntax would be as below,

```

switch (expression)
{
    case value1:
        statement1;
        break;
    case value2:
        statement2;
        break;
    .
    .
    case valueN:
        statementN;
        break;
    default:
        statementDefault;
}

```

The above multi-way branch statement provides an easy way to dispatch execution to different parts of code based on the value of the expression.

[↑ Back to Top](#)

222. What are the conventions to be followed for the usage of switch case

Below are the list of conventions should be taken care,

- i. The expression can be of type either number or string.
- ii. Duplicate values are not allowed for the expression.
- iii. The default statement is optional. If the expression passed to switch does not match with any case value then the statement within default case will be executed.
- iv. The break statement is used inside the switch to terminate a statement sequence.
- v. The break statement is optional. But if it is omitted, the execution will continue on into the next case.

 [Back to Top](#)

223. What are primitive data types

A primitive data type is data that has a primitive value (which has no properties or methods). There are 5 types of primitive data types.

- i. string
- ii. number
- iii. boolean
- iv. null
- v. undefined

 [Back to Top](#)

224. What are the different ways to access object properties

There are 3 possible ways for accessing the property of an object.

- i. **Dot notation:** It uses dot for accessing the properties

`objectName.property`

- i. **Square brackets notation:** It uses square brackets for property access

`objectName["property"]`

- i. **Expression notation:** It uses expression in the square brackets

`objectName[expression]`

 [Back to Top](#)

225. What are the function parameter rules

JavaScript functions follow below rules for parameters,

- i. The function definitions do not specify data types for parameters.
- ii. Do not perform type checking on the passed arguments.
- iii. Do not check the number of arguments received. i.e, The below function follows the above rules,

```
function functionName(parameter1, parameter2, parameter3) {  
    console.log(parameter1); // 1  
}  
functionName(1);
```

 [Back to Top](#)

226. What is an error object

An error object is a built in error object that provides error information when an error occurs. It has two properties: name and message. For example, the below function logs error details,

```
try {  
    greeting("Welcome");  
}  
catch(err) {  
    console.log(err.name + "<br>" + err.message);  
}
```

 [Back to Top](#)

227. When you get a syntax error

A SyntaxError is thrown if you try to evaluate code with a syntax error. For example, the below missing quote for the function parameter throws a syntax error

```
try {  
    eval("greeting('welcome')"); // Missing ' will produce an error  
}  
catch(err) {  
    console.log(err.name);  
}
```

 [Back to Top](#)

228. What are the different error names from error object

There are 6 different types of error names returned from error object,

| Error Name | Description |
|----------------|--|
| EvalError | An error has occurred in the eval() function |
| RangeError | An error has occurred with a number "out of range" |
| ReferenceError | An error due to an illegal reference |
| SyntaxError | An error due to a syntax error |
| TypeError | An error due to a type error |
| URIError | An error due to encodeURI() |

 [Back to Top](#)

229. What are the various statements in error handling

Below are the list of statements used in an error handling,

- i. **try:** This statement is used to test a block of code for errors
- ii. **catch:** This statement is used to handle the error
- iii. **throw:** This statement is used to create custom errors.
- iv. **finally:** This statement is used to execute code after try and catch regardless of the result.

 [Back to Top](#)

230. What are the two types of loops in javascript

- i. **Entry Controlled loops:** In this kind of loop type, the test condition is tested before entering the loop body. For example, For Loop and While Loop comes under this category.
- ii. **Exit Controlled Loops:** In this kind of loop type, the test condition is tested or evaluated at the end of the loop body. i.e, the loop body will execute at least once irrespective of test condition true or false. For example, do-while loop comes under this category.

 [Back to Top](#)

231. What is nodejs

Node.js is a server-side platform built on Chrome's JavaScript runtime for easily building fast and scalable network applications. It is an event-based, non-blocking, asynchronous I/O runtime that uses Google's V8 JavaScript engine and libuv library.

[↑ Back to Top](#)

232. What is an Intl object

The Intl object is the namespace for the ECMAScript Internationalization API, which provides language sensitive string comparison, number formatting, and date and time formatting. It provides access to several constructors and language sensitive functions.

[↑ Back to Top](#)

233. How do you perform language specific date and time formatting

You can use the `Intl.DateTimeFormat` object which is a constructor for objects that enable language-sensitive date and time formatting. Let's see this behavior with an example,

```
var date = new Date(Date.UTC(2019, 07, 03, 0, 0));
console.log(new Intl.DateTimeFormat('en-GB').format(date)); // 07/08/2019
console.log(new Intl.DateTimeFormat('en-AU').format(date)); // 07/08/2019
```

[↑ Back to Top](#)

234. What is an Iterator

An iterator is an object which defines a sequence and a return value upon its termination. It implements the Iterator protocol with a `next()` method which returns an object with two properties: `value` (the next value in the sequence) and `done` (which is true if the last value in the sequence has been consumed).

[↑ Back to Top](#)

235. How does synchronous iteration works

Synchronous iteration was introduced in ES6 and it works with below set of components,

Iterable: It is an object which can be iterated over via a method whose key is `Symbol.iterator`. **Iterator:** It is an object returned by invoking `[Symbol.iterator]()` on an iterable. This iterator object wraps each iterated element in an object and returns it via `next()` method one by one. **IteratorResult:** It is an object returned by `next()` method. The object contains two properties; the `value` property contains an iterated element and the `done` property determines whether the element is the last element or not.

Let's demonstrate synchronous iteration with an array as below,

```
const iterable = ['one', 'two', 'three'];
```

```
const iterator = iterable[Symbol.iterator]();
console.log(iterator.next()); // { value: 'one', done: false }
console.log(iterator.next()); // { value: 'two', done: false }
console.log(iterator.next()); // { value: 'three', done: false }
console.log(iterator.next()); // { value: 'undefined', done: true }
```

[Back to Top](#)

236. What is an event loop

The Event Loop is a queue of callback functions. When an async function executes, the callback function is pushed into the queue. The JavaScript engine doesn't start processing the event loop until the async function has finished executing the code.

Note: It allows Node.js to perform non-blocking I/O operations even though JavaScript is single-threaded.

[Back to Top](#)

237. What is call stack

Call Stack is a data structure for javascript interpreters to keep track of function calls in the program. It has two major actions,

- i. Whenever you call a function for its execution, you are pushing it to the stack.
- ii. Whenever the execution is completed, the function is popped out of the stack.

Let's take an example and it's state representation in a diagram format

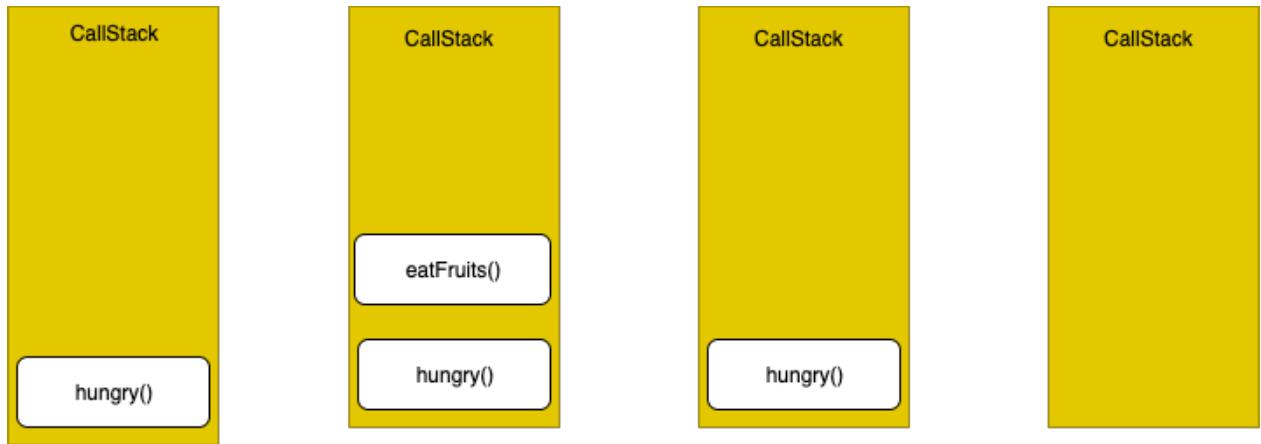
```
function hungry() {
  eatFruits();
}

function eatFruits() {
  return "I'm eating fruits";
}

// Invoke the `hungry` function
hungry();
```

The above code processed in a call stack as below,

- i. Add the `hungry()` function to the call stack list and execute the code.
- ii. Add the `eatFruits()` function to the call stack list and execute the code.
- iii. Delete the `eatFruits()` function from our call stack list.
- iv. Delete the `hungry()` function from the call stack list since there are no items anymore.



[↑ Back to Top](#)

238. What is an event queue

[↑ Back to Top](#)

239. What is a decorator

A decorator is an expression that evaluates to a function and that takes the target, name, and decorator descriptor as arguments. Also, it optionally returns a decorator descriptor to install on the target object. Let's define admin decorator for user class at design time,

```
function admin(isAdmin) {
  return function(target) {
    target.isAdmin = isAdmin;
  }
}

@admin(true)
class User() {
}
console.log(User.isAdmin); //true

@admin(false)
class User() {
}
console.log(User.isAdmin); //false
```

[↑ Back to Top](#)

240. What are the properties of Intl object

Below are the list of properties available on Intl object,

- i. **Collator:** These are the objects that enable language-sensitive string comparison.
- ii. **DateTimeFormat:** These are the objects that enable language-sensitive date and time formatting.

- iii. **ListFormat**: These are the objects that enable language-sensitive list formatting.
- iv. **NumberFormat**: Objects that enable language-sensitive number formatting.
- v. **PluralRules**: Objects that enable plural-sensitive formatting and language-specific rules for plurals.
- vi. **RelativeTimeFormat**: Objects that enable language-sensitive relative time formatting.

 [Back to Top](#)

241. What is an Unary operator

The unary(+) operator is used to convert a variable to a number. If the variable cannot be converted, it will still become a number but with the value NaN. Let's see this behavior in an action.

```
var x = "100";
var y = + x;
console.log(typeof x, typeof y); // string, number

var a = "Hello";
var b = + a;
console.log(typeof a, typeof b, b); // string, number, NaN
```

 [Back to Top](#)

242. How do you sort elements in an array

The sort() method is used to sort the elements of an array in place and returns the sorted array. The example usage would be as below,

```
var months = ["Aug", "Sep", "Jan", "June"];
months.sort();
console.log(months); // ["Aug", "Jan", "June", "Sep"]
```

 [Back to Top](#)

243. What is the purpose of compareFunction while sorting arrays

The compareFunction is used to define the sort order. If omitted, the array elements are converted to strings, then sorted according to each character's Unicode code point value. Let's take an example to see the usage of compareFunction,

```
let numbers = [1, 2, 5, 3, 4];
numbers.sort((a, b) => b - a);
console.log(numbers); // [5, 4, 3, 2, 1]
```

[↑ Back to Top](#)

244. How do you reversing an array

You can use the `reverse()` method to reverse the elements in an array. This method is useful to sort an array in descending order. Let's see the usage of `reverse()` method in an example,

```
let numbers = [1, 2, 5, 3, 4];
numbers.sort((a, b) => b - a);
numbers.reverse();
console.log(numbers); // [1, 2, 3, 4 ,5]
```

[↑ Back to Top](#)

245. How do you find min and max value in an array

You can use `Math.min` and `Math.max` methods on array variables to find the minimum and maximum elements within an array. Let's create two functions to find the min and max value with in an array,

```
var marks = [50, 20, 70, 60, 45, 30];
function findMin(arr) {
    return Math.min(...arr);
}
function findMax(arr) {
    return Math.max(...arr);
}

console.log(findMin(marks));
console.log(findMax(marks));
```

[↑ Back to Top](#)

246. How do you find min and max values without Math functions

You can write functions which loop through an array comparing each value with the lowest value or highest value to find the min and max values. Let's create those functions to find min and max values,

```
var marks = [50, 20, 70, 60, 45, 30];
function findMin(arr) {
    var length = arr.length
    var min = Infinity;
    while (length--) {
        if (arr[length] < min) {
            min = arr[length];
        }
    }
    return min;
}

function findMax(arr) {
    var length = arr.length
    var max = -Infinity;
    while (length--) {
        if (arr[length] > max) {
            max = arr[length];
        }
    }
    return max;
}
```

```

        }
    }

    return min;
}

function findMax(arr) {
    var length = arr.length
    var max = -Infinity;
    while (len--) {
        if (arr[length] > max) {
            max = arr[length];
        }
    }
    return max;
}

console.log(findMin(marks));
console.log(findMax(marks));

```

[↑ Back to Top](#)

247. What is an empty statement and purpose of it

The empty statement is a semicolon (;) indicating that no statement will be executed, even if JavaScript syntax requires one. Since there is no action with an empty statement you might think that its usage is quite less, but the empty statement is occasionally useful when you want to create a loop that has an empty body. For example, you can initialize an array with zero values as below,

```
// Initialize an array a
for(int i=0; i < a.length; a[i++] = 0) ;
```

[↑ Back to Top](#)

248. How do you get metadata of a module

You can use the `import.meta` object which is a meta-property exposing context-specific meta data to a JavaScript module. It contains information about the current module, such as the module's URL. In browsers, you might get different meta data than NodeJS.

```
<script type="module" src="welcome-module.js"></script>
console.log(import.meta); // { url: "file:///home/user/welcome-module.js" }
```

[↑ Back to Top](#)

249. What is a comma operator

The comma operator is used to evaluate each of its operands from left to right and returns the value of the last operand. This is totally different from comma usage within arrays, objects, and function arguments and parameters. For example, the usage for numeric expressions would be as below,

```
var x = 1;  
x = (x++, x);  
  
console.log(x); // 2
```

 [Back to Top](#)

250. What is the advantage of a comma operator

It is normally used to include multiple expressions in a location that requires a single expression. One of the common usages of this comma operator is to supply multiple parameters in a `for` loop. For example, the below for loop uses multiple expressions in a single location using comma operator,

```
for (var a = 0, b = 10; a <= 10; a++, b--)
```

You can also use the comma operator in a return statement where it processes before returning.

```
function myFunction() {  
    var a = 1;  
    return (a += 10, a); // 11  
}
```

 [Back to Top](#)

251. What is typescript

TypeScript is a typed superset of JavaScript created by Microsoft that adds optional types, classes, async/await, and many other features, and compiles to plain JavaScript. Angular built entirely in TypeScript and used as a primary language. You can install it globally as

```
npm install -g typescript
```

Let's see a simple example of TypeScript usage,

```
function greeting(name: string): string {  
    return "Hello, " + name;
```

```

    }

let user = "Sudheer";

console.log(greeting(user));

```

The greeting method allows only string type as argument.

[↑ Back to Top](#)

252. What are the differences between javascript and typescript

Below are the list of differences between javascript and typescript,

| feature | typescript | javascript |
|---------------------|---------------------------------------|---|
| Language paradigm | Object oriented programming language | Scripting language |
| Typing support | Supports static typing | It has dynamic typing |
| Modules | Supported | Not supported |
| Interface | It has interfaces concept | Doesn't support interfaces |
| Optional parameters | Functions support optional parameters | No support of optional parameters for functions |

[↑ Back to Top](#)

253. What are the advantages of typescript over javascript

Below are some of the advantages of typescript over javascript,

- i. TypeScript is able to find compile time errors at the development time only and it makes sure less runtime errors. Whereas javascript is an interpreted language.
- ii. TypeScript is strongly-typed or supports static typing which allows for checking type correctness at compile time. This is not available in javascript.
- iii. TypeScript compiler can compile the .ts files into ES3,ES4 and ES5 unlike ES features of javascript which may not be supported in some browsers.

[↑ Back to Top](#)

254. What is an object initializer

An object initializer is an expression that describes the initialization of an Object. The syntax for this expression is represented as a comma-delimited list of zero or more pairs of property names and associated values of an object, enclosed in curly braces ({}). This is also known as literal notation. It is one of the ways to create an object.

```
var initObject = {a: 'John', b: 50, c: {}};

console.log(initObject.a); // John
```

 [Back to Top](#)

255. What is a constructor method

The constructor method is a special method for creating and initializing an object created within a class. If you do not specify a constructor method, a default constructor is used. The example usage of constructor would be as below,

```
class Employee {
  constructor() {
    this.name = "John";
  }
}

var employeeObject = new Employee();

console.log(employeeObject.name); // John
```

 [Back to Top](#)

256. What happens if you write constructor more than once in a class

The "constructor" in a class is a special method and it should be defined only once in a class. i.e, If you write a constructor method more than once in a class it will throw a `SyntaxError` error.

```

class Employee {
    constructor() {
        this.name = "John";
    }
    constructor() { // Uncaught SyntaxError: A class may only have one construct
        this.age = 30;
    }
}

var employeeObject = new Employee();

console.log(employeeObject.name);

```

[↑ Back to Top](#)

257. How do you call the constructor of a parent class

You can use the `super` keyword to call the constructor of a parent class. Remember that `super()` must be called before using 'this' reference. Otherwise it will cause a reference error. Let's see the usage of it,

```

class Square extends Rectangle {
    constructor(length) {
        super(length, length);
        this.name = 'Square';
    }

    get area() {
        return this.width * this.height;
    }

    set area(value) {
        this.area = value;
    }
}

```

[↑ Back to Top](#)

258. How do you get the prototype of an object

You can use the `Object.getPrototypeOf(obj)` method to return the prototype of the specified object. i.e. The value of the internal `prototype` property. If there are no inherited properties then `null` value is returned.

```
const newPrototype = {};
const newObj = Object.create(newPrototype);

console.log(Object.getPrototypeOf(newObj) === newPrototype); // true
```

[↑ Back to Top](#)

259. What happens If I pass string type for getPrototypeOf method

In ES5, it will throw a `TypeError` exception if the `obj` parameter isn't an object. Whereas in ES2015, the parameter will be coerced to an `Object`.

```
// ES5
Object.getPrototypeOf('James'); // TypeError: "James" is not an object
// ES2015
Object.getPrototypeOf('James'); // String.prototype
```

[↑ Back to Top](#)

260. How do you set prototype of one object to another

You can use the `Object.setPrototypeOf()` method that sets the prototype (i.e., the internal `Prototype` property) of a specified object to another object or null. For example, if you want to set prototype of a square object to rectangle object would be as follows,

```
Object.setPrototypeOf(Square.prototype, Rectangle.prototype);
Object.setPrototypeOf({}, null);
```

[↑ Back to Top](#)

261. How do you check whether an object can be extendable or not

The `Object.isExtensible()` method is used to determine if an object is extendable or not. i.e, Whether it can have new properties added to it or not.

```
const newObj = {};
console.log(Object.isExtensible(newObj)); //true
```

Note: By default, all the objects are extendable. i.e, The new properties can be added or modified.

[↑ Back to Top](#)

262. How do you prevent an object to extend

The `Object.preventExtensions()` method is used to prevent new properties from ever being added to an object. In other words, it prevents future extensions to the object. Let's see the usage of this property,

```
const newObject = {};
Object.preventExtensions(newObject); // NOT extendable

try {
  Object.defineProperty(newObject, 'newProperty', { // Adding new property
    value: 100
  });
} catch (e) {
  console.log(e); // TypeError: Cannot define property newProperty, object is not
}
```

[↑ Back to Top](#)

263. What are the different ways to make an object non-extensible

You can mark an object non-extensible in 3 ways,

- i. `Object.preventExtensions`
- ii. `Object.seal`
- iii. `Object.freeze`

```
var newObject = {};

Object.preventExtensions(newObject); // Prevent objects are non-extensible
Object.isExtensible(newObject); // false

var sealedObject = Object.seal({}); // Sealed objects are non-extensible
Object.isExtensible(sealedObject); // false

var frozenObject = Object.freeze({}); // Frozen objects are non-extensible
Object.isExtensible(frozenObject); // false
```

[↑ Back to Top](#)

264. How do you define multiple properties on an object

The `Object.defineProperties()` method is used to define new or modify existing properties directly on an object and returning the object. Let's define multiple properties on an empty object,

```
const newObject = {};
```

```
Object.defineProperties(newObject, {  
    newProperty1: {  
        value: 'John',  
        writable: true  
    },  
    newProperty2: {}  
});
```

[↑ Back to Top](#)

265. What is MEAN in javascript

The MEAN (MongoDB, Express, AngularJS, and Node.js) stack is the most popular open-source JavaScript software tech stack available for building dynamic web apps where you can write both the server-side and client-side halves of the web project entirely in JavaScript.

[↑ Back to Top](#)

266. What Is Obfuscation in javascript

Obfuscation is the deliberate act of creating obfuscated javascript code(i.e, source or machine code) that is difficult for humans to understand. It is something similar to encryption, but a machine can understand the code and execute it. Let's see the below function before Obfuscation,

```
function greeting() {  
    console.log('Hello, welcome to JS world');  
}
```

And after the code Obfuscation, it would be appeared as below,

```
eval(function(p,a,c,k,e,d){e=function(c){return c};if(!''.replace(/\//,String)){whi
```

[↑ Back to Top](#)

267. Why do you need Obfuscation

Below are the few reasons for Obfuscation,

- i. The Code size will be reduced. So data transfers between server and client will be fast.
- ii. It hides the business logic from outside world and protects the code from others
- iii. Reverse engineering is highly difficult
- iv. The download time will be reduced

[↑ Back to Top](#)

268. What is Minification

Minification is the process of removing all unnecessary characters(empty spaces are removed) and variables will be renamed without changing it's functionality. It is also a type of obfuscation .

[↑ Back to Top](#)

269. What are the advantages of minification

Normally it is recommended to use minification for heavy traffic and intensive requirements of resources. It reduces file sizes with below benefits,

- i. Decreases loading times of a web page
- ii. Saves bandwidth usages

[↑ Back to Top](#)

270. What are the differences between Obfuscation and Encryption

Below are the main differences between Obfuscation and Encryption,

| Feature | Obfuscation | Encryption |
|--------------------|---|---|
| Definition | Changing the form of any data in any other form | Changing the form of information to an unreadable format by using a key |
| A key to decode | It can be decoded without any key | It is required |
| Target data format | It will be converted to a complex form | Converted into an unreadable format |

[↑ Back to Top](#)

271. What are the common tools used for minification

There are many online/offline tools to minify the javascript files,

- i. Google's Closure Compiler
- ii. UglifyJS2
- iii. jsmin
- iv. javascript-minifier.com/
- v. prettydiff.com

[↑ Back to Top](#)

272. How do you perform form validation using javascript

JavaScript can be used to perform HTML form validation. For example, if the form field is empty, the function needs to notify, and return false, to prevent the form being submitted. Lets' perform user login in an html form,

```
<form name="myForm" onsubmit="return validateForm()" method="post">
User name: <input type="text" name="uname">
<input type="submit" value="Submit">
</form>
```

And the validation on user login is below,

```
function validateForm() {
    var x = document.forms["myForm"]["uname"].value;
    if (x == "") {
        alert("The username shouldn't be empty");
        return false;
    }
}
```

[↑ Back to Top](#)

273. How do you perform form validation without javascript

You can perform HTML form validation automatically without using javascript. The validation enabled by applying the `required` attribute to prevent form submission when the input is empty.

```
<form method="post">
    <input type="text" name="uname" required>
    <input type="submit" value="Submit">
</form>
```

Note: Automatic form validation does not work in Internet Explorer 9 or earlier.

[↑ Back to Top](#)

274. What are the DOM methods available for constraint validation

The below DOM methods are available for constraint validation on an invalid input,

- i. `checkValidity()`: It returns true if an input element contains valid data.
- ii. `setCustomValidity()`: It is used to set the `validationMessage` property of an input

element. Let's take an user login form with DOM validations

```
function myFunction() {  
    var userName = document.getElementById("uname");  
    if (!userName.checkValidity()) {  
        document.getElementById("message").innerHTML = userName.validationMessage;  
    } else {  
        document.getElementById("message").innerHTML = "Entered a valid username";  
    }  
}
```

 [Back to Top](#)

275. What are the available constraint validation DOM properties

Below are the list of some of the constraint validation DOM properties available,

- i. validity: It provides a list of boolean properties related to the validity of an input element.
- ii. validationMessage: It displays the message when the validity is false.
- iii. willValidate: It indicates if an input element will be validated or not.

 [Back to Top](#)

276. What are the list of validity properties

The validity property of an input element provides a set of properties related to the validity of data.

- i. customError: It returns true, if a custom validity message is set.
- ii. patternMismatch: It returns true, if an element's value does not match its pattern attribute.
- iii. rangeOverflow: It returns true, if an element's value is greater than its max attribute.
- iv. rangeUnderflow: It returns true, if an element's value is less than its min attribute.
- v. stepMismatch: It returns true, if an element's value is invalid according to step attribute.
- vi. tooLong: It returns true, if an element's value exceeds its maxLength attribute.
- vii. typeMismatch: It returns true, if an element's value is invalid according to type attribute.
- viii. valueMissing: It returns true, if an element with a required attribute has no value.
- ix. valid: It returns true, if an element's value is valid.

 [Back to Top](#)

277. Give an example usage of rangeOverflow property

If an element's value is greater than its max attribute then rangeOverflow property returns true. For example, the below form submission throws an error if the value is more than 100,

```
<input id="age" type="number" max="100">
<button onclick="myOverflowFunction()">OK</button>

function myOverflowFunction() {
  if (document.getElementById("age").validity.rangeOverflow) {
    alert("The mentioned age is not allowed");
  }
}
```

[↑ Back to Top](#)

278. Is enums feature available in javascript

No, javascript does not natively support enums. But there are different kinds of solutions to simulate them even though they may not provide exact equivalents. For example, you can use freeze or seal on object,

```
var DaysEnum = Object.freeze({"monday":1, "tuesday":2, "wednesday":3, ...})
```

[↑ Back to Top](#)

279. What is an enum

An enum is a type restricting variables to one value from a predefined set of constants. JavaScript has no enums but typescript provides built-in enum support.

```
enum Color {
  RED, GREEN, BLUE
}
```

[↑ Back to Top](#)

280. How do you list all properties of an object

You can use the `Object.getOwnPropertyNames()` method which returns an array of all properties found directly in a given object. Let's the usage of it in an example,

```
const newObject = {
  a: 1,
  b: 2,
```

```
    c: 3
};

console.log(Object.getOwnPropertyNames(newObject)); ["a", "b", "c"]
```

[↑ Back to Top](#)

281. How do you get property descriptors of an object

You can use the `Object.getOwnPropertyDescriptors()` method which returns all own property descriptors of a given object. The example usage of this method is below,

```
const newObject = {
  a: 1,
  b: 2,
  c: 3
};
const descriptorsObject = Object.getOwnPropertyDescriptors(newObject);
console.log(descriptorsObject.a.writable); //true
console.log(descriptorsObject.a.configurable); //true
console.log(descriptorsObject.a.enumerable); //true
console.log(descriptorsObject.a.value); // 1
```

[↑ Back to Top](#)

282. What are the attributes provided by a property descriptor

A property descriptor is a record which has the following attributes

- i. value: The value associated with the property
- ii. writable: Determines whether the value associated with the property can be changed or not
- iii. configurable: Returns true if the type of this property descriptor can be changed and if the property can be deleted from the corresponding object.
- iv. enumerable: Determines whether the property appears during enumeration of the properties on the corresponding object or not.
- v. set: A function which serves as a setter for the property
- vi. get: A function which serves as a getter for the property

[↑ Back to Top](#)

283. How do you extend classes

The `extends` keyword is used in class declarations/expressions to create a class which is a child of another class. It can be used to subclass custom classes as well as built-in objects. The syntax would be as below,

```
class ChildClass extends ParentClass { ... }
```

Let's take an example of Square subclass from Polygon parent class,

```
class Square extends Rectangle {
  constructor(length) {
    super(length, length);
    this.name = 'Square';
  }

  get area() {
    return this.width * this.height;
  }

  set area(value) {
    this.area = value;
  }
}
```

 [Back to Top](#)

284. How do I modify the url without reloading the page

The `window.location.url` property will be helpful to modify the url but it reloads the page. HTML5 introduced the `history.pushState()` and `history.replaceState()` methods, which allow you to add and modify history entries, respectively. For example, you can use `pushState` as below,

```
window.history.pushState('page2', 'Title', '/page2.html');
```

 [Back to Top](#)

285. How do you check whether an array includes a particular value or not

The `Array#includes()` method is used to determine whether an array includes a particular value among its entries by returning either true or false. Let's see an example to find an element(numeric and string) within an array.

```
var numericArray = [1, 2, 3, 4];
console.log(numericArray.includes(3)); // true

var stringArray = ['green', 'yellow', 'blue'];
console.log(stringArray.includes('blue'));
```

[↑ Back to Top](#)

286. How do you compare scalar arrays

You can use length and every method of arrays to compare two scalar(compared directly using ===) arrays. The combination of these expressions can give the expected result,

```
const arrayFirst = [1,2,3,4,5];
const arraySecond = [1,2,3,4,5];
console.log(arrayFirst.length === arraySecond.length && arrayFirst.every((value, i
```

If you would like to compare arrays irrespective of order then you should sort them before,

```
const arrayFirst = [2,3,1,4,5];
const arraySecond = [1,2,3,4,5];
console.log(arrayFirst.length === arraySecond.length && arrayFirst.sort().every((v
```

[↑ Back to Top](#)

287. How to get the value from get parameters

The new URL() object accepts the url string and searchParams property of this object can be used to access the get parameters. Remember that you may need to use polyfill or window.location to access the URL in older browsers(including IE).

```
let urlString = "http://www.some-domain.com/about.html?x=1&y=2&z=3"; //window.location
let url = new URL(urlString);
let parameterZ = url.searchParams.get("z");
console.log(parameterZ); // 3
```

[↑ Back to Top](#)

288. How do you print numbers with commas as thousand separators

You can use the Number.prototype.toLocaleString() method which returns a string with a language-sensitive representation such as thousand separator,currency etc of this number.

```
function convertToThousandFormat(x){
    return x.toLocaleString(); // 12,345.679
}

console.log(convertToThousandFormat(12345.6789));
```

[↑ Back to Top](#)

289. What is the difference between java and javascript

Both are totally unrelated programming languages and no relation between them. Java is statically typed, compiled, runs on its own VM. Whereas Javascript is dynamically typed, interpreted, and runs in a browser and nodejs environments. Let's see the major differences in a tabular format,

| Feature | Java | JavaScript |
|-------------|--------------------------------|---|
| Typed | It's a strongly typed language | It's a dynamic typed language |
| Paradigm | Object oriented programming | Prototype based programming |
| Scoping | Block scoped | Function-scoped |
| Concurrency | Thread based | event based |
| Memory | Uses more memory | Uses less memory. Hence it will be used for web pages |

[↑ Back to Top](#)

290. Is javascript supports namespace

JavaScript doesn't support namespace by default. So if you create any element(function, method, object, variable) then it becomes global and pollutes the global namespace. Let's take an example of defining two functions without any namespace,

```
function func1() {  
    console.log("This is a first definition");  
  
}  
function func1() {  
    console.log("This is a second definition");  
}  
func1(); // This is a second definition
```

It always calls the second function definition. In this case, namespace will solve the name collision problem.

[↑ Back to Top](#)

291. How do you declare namespace

Even though JavaScript lacks namespaces, we can use Objects , IIFE to create namespaces.

i. **Using Object Literal Notation:** Let's wrap variables and functions inside an Object literal which acts as a namespace. After that you can access them using object notation

```
var namespaceOne = {
    function func1() {
        console.log("This is a first definition");
    }
}
var namespaceTwo = {
    function func1() {
        console.log("This is a second definition");
    }
}
namespaceOne.func1(); // This is a first definition
namespaceTwo.func1(); // This is a second definition
```

i. **Using IIFE (Immediately invoked function expression):** The outer pair of parentheses of IIFE creates a local scope for all the code inside of it and makes the anonymous function a function expression. Due to that, you can create the same function in two different function expressions to act as a namespace.

```
(function() {
    function fun1(){
        console.log("This is a first definition");
    } fun1();
}());

(function() {
    function fun1(){
        console.log("This is a second definition");
    } fun1();
}());
```

i. **Using a block and a let/const declaration:** In ECMAScript 6, you can simply use a block and a let declaration to restrict the scope of a variable to a block.

```
{
let myFunction= function fun1(){
console.log("This is a first definition");
}
myFunction();
}
//myFunction(): ReferenceError: myFunction is not defined.

{
let myFunction= function fun1(){
console.log("This is a second definition");
```

```
    }
    myFunction();
}
//myFunction(): ReferenceError: myFunction is not defined.
```

[↑ Back to Top](#)

292. How do you invoke javascript code in an iframe from parent page

Initially iFrame needs to be accessed using either `document.getElementById` or `window.frames`. After that `contentWindow` property of iFrame gives the access for `targetFunction`

```
document.getElementById('targetFrame').contentWindow.targetFunction();
window.frames[0].frameElement.contentWindow.targetFunction(); // Accessing iframe
```

[↑ Back to Top](#)

293. How do get the timezone offset from date

You can use the `getTimezoneOffset` method of the date object. This method returns the time zone difference, in minutes, from current locale (host system settings) to UTC

```
var offset = new Date().getTimezoneOffset();
console.log(offset); // -480
```

[↑ Back to Top](#)

294. How do you load CSS and JS files dynamically

You can create both link and script elements in the DOM and append them as child to head tag. Let's create a function to add script and style resources as below,

```

function loadAssets(filename, filetype) {
    if (filetype == "css") { // External CSS file
        var fileReference = document.createElement("link");
        fileReference.setAttribute("rel", "stylesheet");
        fileReference.setAttribute("type", "text/css");
        fileReference.setAttribute("href", filename);
    } else if (filetype == "js") { // External JavaScript file
        var fileReference = document.createElement('script');
        fileReference.setAttribute("type", "text/javascript");
        fileReference.setAttribute("src", filename);
    }
    if (typeof fileReference != "undefined")
        document.getElementsByTagName("head")[0].appendChild(fileReference)
}

```

[↑ Back to Top](#)

295. What are the different methods to find HTML elements in DOM

If you want to access any element in an HTML page, you need to start with accessing the document object. Later you can use any of the below methods to find the HTML element,

- i. `document.getElementById(id)`: It finds an element by Id
- ii. `document.getElementsByTagName(name)`: It finds an element by tag name
- iii. `document.getElementsByClassName(name)`: It finds an element by class name

[↑ Back to Top](#)

296. What is jQuery

jQuery is a popular cross-browser JavaScript library that provides Document Object Model (DOM) traversal, event handling, animations and AJAX interactions by minimizing the discrepancies across browsers. It is widely famous with its philosophy of "Write less, do more". For example, you can display welcome message on the page load using jQuery as below,

```

$(document).ready(function(){ // It selects the document and apply the function or
    alert('Welcome to jQuery world');
});

```

Note: You can download it from jquery's official site or install it from CDNs, like google.

[↑ Back to Top](#)

297. What is V8 JavaScript engine

V8 is an open source high-performance JavaScript engine used by the Google Chrome browser, written in C++. It is also being used in the nodejs project. It implements ECMAScript and WebAssembly, and runs on Windows 7 or later, macOS 10.12+, and Linux systems that use x64, IA-32, ARM, or MIPS processors. **Note:** It can run standalone, or can be embedded into any C++ application.

[↑ Back to Top](#)

298. Why do we call javascript as dynamic language

JavaScript is a loosely typed or a dynamic language because variables in JavaScript are not directly associated with any particular value type, and any variable can be assigned/reassigned with values of all types.

```
let age = 50;      // age is a number now
age = 'old'; // age is a string now
age = true; // age is a boolean
```

[↑ Back to Top](#)

299. What is a void operator

The `void` operator evaluates the given expression and then returns undefined(i.e, without returning value). The syntax would be as below,

```
void (expression)
void expression
```

Let's display a message without any redirection or reload

```
<a href="javascript:void(alert('Welcome to JS world'))">Click here to see a message
```

Note: This operator is often used to obtain the undefined primitive value, using "void(0)".

[↑ Back to Top](#)

300. How to set the cursor to wait

The cursor can be set to wait in JavaScript by using the property "cursor". Let's perform this behavior on page load using the below function.

```
function myFunction() {  
    window.document.body.style.cursor = "wait";  
}
```

and this function invoked on page load

```
<body onload="myFunction()">
```

 [Back to Top](#)

301. How do you create an infinite loop

You can create infinite loops using for and while loops without using any expressions. The for loop construct or syntax is better approach in terms of ESLint and code optimizer tools,

```
for (;;) {}  
while(true) {  
}
```

 [Back to Top](#)

302. Why do you need to avoid with statement

JavaScript's with statement was intended to provide a shorthand for writing recurring accesses to objects. So it can help reduce file size by reducing the need to repeat a lengthy object reference without performance penalty. Let's take an example where it is used to avoid redundancy when accessing an object several times.

```
a.b.c.greeting = 'welcome';  
a.b.c.age = 32;
```

Using with it turns this into:

```
with(a.b.c) {  
    greeting = "welcome";  
    age = 32;  
}
```

But this with statement creates performance problems since one cannot predict whether an argument will refer to a real variable or to a property inside the with argument.

[↑ Back to Top](#)

303. What is the output of below for loops

```
for (var i = 0; i < 4; i++) { // global scope
  setTimeout(() => console.log(i));
}

for (let i = 0; i < 4; i++) { // block scope
  setTimeout(() => console.log(i));
}
```

The output of the above for loops is 4 4 4 4 and 0 1 2 3 **Explanation:** Due to the event queue/loop of javascript, the `setTimeout` callback function is called after the loop has been executed. Since the variable `i` is declared with the `var` keyword it became a global variable and the value was equal to 4 using iteration when the time `setTimeout` function is invoked. Hence, the output of the first loop is 4 4 4 4 . Whereas in the second loop, the variable `i` is declared as the `let` keyword it becomes a block scoped variable and it holds a new value(0, 1 ,2 3) for each iteration. Hence, the output of the first loop is 0 1 2 3 .

[↑ Back to Top](#)

304. List down some of the features of ES6

Below are the list of some new features of ES6,

- i. Support for constants or immutable variables
- ii. Block-scope support for variables, constants and functions
- iii. Arrow functions
- iv. Default parameters
- v. Rest and Spread Parameters
- vi. Template Literals
- vii. Multi-line Strings
- viii. Destructuring Assignment
- ix. Enhanced Object Literals
- x. Promises
- xi. Classes
- xii. Modules

[↑ Back to Top](#)

305. What is ES6

ES6 is the sixth edition of the javascript language and it was released in June 2015. It was initially known as ECMAScript 6 (ES6) and later renamed to ECMAScript 2015. Almost all the modern browsers support ES6 but for the old browsers there are many transpilers, like Babel.js etc.

[↑ Back to Top](#)

306. Can I redeclare let and const variables

No, you cannot redeclare let and const variables. If you do, it throws below error

```
Uncaught SyntaxError: Identifier 'someVariable' has already been declared
```

Explanation: The variable declaration with `var` keyword refers to a function scope and the variable is treated as if it were declared at the top of the enclosing scope due to hoisting feature. So all the multiple declarations contributing to the same hoisted variable without any error. Let's take an example of re-declaring variables in the same scope for both `var` and `let/const` variables.

```
var name = 'John';
function myFunc() {
    var name = 'Nick';
    var name = 'Abraham'; // Re-assigned in the same function block
    alert(name); // Abraham
}
myFunc();
alert(name); // John
```

The block-scoped multi-declaration throws syntax error,

```
let name = 'John';
function myFunc() {
    let name = 'Nick';
    let name = 'Abraham'; // Uncaught SyntaxError: Identifier 'name' has already t
    alert(name);
}

myFunc();
alert(name);
```

[↑ Back to Top](#)

307. Is const variable makes the value immutable

No, the `const` variable doesn't make the value immutable. But it disallows subsequent assignments(i.e, You can declare with assignment but can't assign another value later)

```
const userList = [];
userList.push('John'); // Can mutate even though it can't re-assign
console.log(userList); // ['John']
```

[↑ Back to Top](#)

308. What are default parameters

In ES5, we need to depend on logical OR operators to handle default values of function parameters. Whereas in ES6, Default function parameters feature allows parameters to be initialized with default values if no value or undefined is passed. Let's compare the behavior with an examples,

```
//ES5
var calculateArea = function(height, width) {
    height = height || 50;
    width = width || 60;

    return width * height;
}
console.log(calculateArea()); //300
```

The default parameters makes the initialization more simpler,

```
//ES6
var calculateArea = function(height = 50, width = 60) {
    return width * height;
}

console.log(calculateArea()); //300
```

[↑ Back to Top](#)

309. What are template literals

Template literals or template strings are string literals allowing embedded expressions. These are enclosed by the back-tick (`) character instead of double or single quotes. In ES6, this feature enables using dynamic expressions as below,

```
var greeting = `Welcome to JS World, Mr. ${firstName} ${lastName}.`
```

In ES5, you need break string like below,

```
var greeting = 'Welcome to JS World, Mr. ' + firstName + ' ' + lastName.'
```

Note: You can use multi-line strings and string interpolation features with template literals.

[↑ Back to Top](#)

310. How do you write multi-line strings in template literals

In ES5, you would have to use newline escape characters('\n') and concatenation symbols(+) in order to get multi-line strings.

```
console.log('This is string sentence 1\n' +
'This is string sentence 2');
```

Whereas in ES6, You don't need to mention any newline sequence character,

```
console.log(`This is string sentence
'This is string sentence 2`);
```

[↑ Back to Top](#)

311. What are nesting templates

The nesting template is a feature supported within template literals syntax to allow inner backticks inside a placeholder \${ } within the template. For example, the below nesting template is used to display the icons based on user permissions whereas outer template checks for platform type,

```
const iconStyles = `icon ${ isMobilePlatform() ? '' :
`icon-${user.isAuthorized ? 'submit' : 'disabled'} }`;
```

You can write the above use case without nesting template features as well. However, the nesting template feature is more compact and readable.

```
//Without nesting templates
const iconStyles = `icon ${ isMobilePlatform() ? '' :
(user.isAuthorized ? 'icon-submit' : 'icon-disabled') }`;
```

[↑ Back to Top](#)

312. What are tagged templates

Tagged templates are the advanced form of templates in which tags allow you to parse template literals with a function. The tag function accepts the first parameter as an array of strings and remaining parameters as expressions. This function can also return manipulated strings based on parameters. Let's see the usage of this tagged template behavior of an IT professional skill set in an organization,

```
var user1 = 'John';
var skill1 = 'JavaScript';
var experience1 = 15;

var user2 = 'Kane';
var skill2 = 'JavaScript';
var experience2 = 5;

function myInfoTag(strings, userExp, experienceExp, skillExp) {
    var str0 = strings[0]; // "Mr/Ms. "
    var str1 = strings[1]; // " is a/an "
    var str2 = strings[2]; // "in"

    var expertiseStr;
    if (experienceExp > 10){
        expertiseStr = 'expert developer';
    } else if(skillExp > 5 && skillExp <= 10) {
        expertiseStr = 'senior developer';
    } else {
        expertiseStr = 'junior developer';
    }

    return ${str0}${userExp}${str1}${expertiseStr}${str2}${skillExp};
}

var output1 = myInfoTag`Mr/Ms. ${ user1 } is a/an ${ experience1 } in ${skill1}`;
var output2 = myInfoTag`Mr/Ms. ${ user2 } is a/an ${ experience2 } in ${skill2}`;

console.log(output1); // Mr/Ms. John is a/an expert developer in JavaScript
console.log(output2); // Mr/Ms. Kane is a/an junior developer in JavaScript
```

 [Back to Top](#)

313. What are raw strings

ES6 provides a raw strings feature using the `String.raw()` method which is used to get the raw string form of template strings. This feature allows you to access the raw strings as they were entered, without processing escape sequences. For example, the usage would be as below,

```
var calculationString = String.raw `The sum of numbers is \n${1+2+3+4}!`;
console.log(calculationString); // The sum of numbers is 10
```

If you don't use raw strings, the newline character sequence will be processed by displaying the output in multiple lines

```
var calculationString = `The sum of numbers is \n${1+2+3+4}!`;
console.log(calculationString);
// The sum of numbers is
// 10
```

Also, the raw property is available on the first argument to the tag function

```
function tag(strings) {
  console.log(strings.raw[0]);
}
```

 [Back to Top](#)

314. What is destructuring assignment

The destructuring assignment is a JavaScript expression that makes it possible to unpack values from arrays or properties from objects into distinct variables. Let's get the month values from an array using destructuring assignment

```
var [one, two, three] = ['JAN', 'FEB', 'MARCH'];

console.log(one); // "JAN"
console.log(two); // "FEB"
console.log(three); // "MARCH"
```

and you can get user properties of an object using destructuring assignment,

```
var {name, age} = {name: 'John', age: 32};

console.log(name); // John
console.log(age); // 32
```

 [Back to Top](#)

315. What are default values in destructuring assignment

A variable can be assigned a default value when the value unpacked from the array or object is undefined during destructuring assignment. It helps to avoid setting default values separately for each assignment. Let's take an example for both arrays and object use cases,

Arrays destructuring:

```
var x, y, z;  
  
[x=2, y=4, z=6] = [10];  
console.log(x); // 10  
console.log(y); // 4  
console.log(z); // 6
```

Objects destructuring:

```
var {x=2, y=4, z=6} = {x: 10};  
  
console.log(x); // 10  
console.log(y); // 4  
console.log(z); // 6
```

 [Back to Top](#)

316. How do you swap variables in destructuring assignment

If you don't use destructuring assignment, swapping two values requires a temporary variable. Whereas using a destructuring feature, two variable values can be swapped in one destructuring expression. Let's swap two number variables in array destructuring assignment,

```
var x = 10, y = 20;  
  
[x, y] = [y, x];  
console.log(x); // 20  
console.log(y); // 10
```

 [Back to Top](#)

317. What are enhanced object literals

Object literals make it easy to quickly create objects with properties inside the curly braces. For example, it provides shorter syntax for common object property definition as below.

```
//ES6
var x = 10, y = 20
obj = { x, y }
console.log(obj); // {x: 10, y:20}
//ES5
var x = 10, y = 20
obj = { x : x, y : y}
console.log(obj); // {x: 10, y:20}
```

[↑ Back to Top](#)

318. What are dynamic imports

The dynamic imports using `import()` function syntax allows us to load modules on demand by using promises or the `async/await` syntax. Currently this feature is in [stage4 proposal](#). The main advantage of dynamic imports is reduction of our bundle's sizes, the size/payload response of our requests and overall improvements in the user experience. The syntax of dynamic imports would be as below,

```
import('./Module').then(Module => Module.method());
```

[↑ Back to Top](#)

319. What are the use cases for dynamic imports

Below are some of the use cases of using dynamic imports over static imports,

- i. Import a module on-demand or conditionally. For example, if you want to load a polyfill on legacy browser

```
if (isLegacyBrowser()) {
  import(...)
  .then(...);
}
```

- i. Compute the module specifier at runtime. For example, you can use it for internationalization.

```
import(`messages_${getLocale()}.js`).then(...);
```

- i. Import a module from within a regular script instead a module.

[↑ Back to Top](#)

320. What are typed arrays

Typed arrays are array-like objects from ECMAScript 6 API for handling binary data. JavaScript provides 8 Typed array types,

- i. Int8Array: An array of 8-bit signed integers
- ii. Int16Array: An array of 16-bit signed integers
- iii. Int32Array: An array of 32-bit signed integers
- iv. Uint8Array: An array of 8-bit unsigned integers
- v. Uint16Array: An array of 16-bit unsigned integers
- vi. Uint32Array: An array of 32-bit unsigned integers
- vii. Float32Array: An array of 32-bit floating point numbers
- viii. Float64Array: An array of 64-bit floating point numbers

For example, you can create an array of 8-bit signed integers as below

```
const a = new Int8Array();
// You can pre-allocate n bytes
const bytes = 1024
const a = new Int8Array(bytes)
```

 [Back to Top](#)

321. What are the advantages of module loaders

The module loaders provides the below features,

- i. Dynamic loading
- ii. State isolation
- iii. Global namespace isolation
- iv. Compilation hooks
- v. Nested virtualization

 [Back to Top](#)

322. What is collation

Collation is used for sorting a set of strings and searching within a set of strings. It is parameterized by locale and aware of Unicode. Let's take comparison and sorting features,

- i. Comparison:

```
var list = [ "ä", "a", "z" ]; // In German, "ä" sorts with "a" Whereas in Swedish  
var l10nDE = new Intl.Collator("de");  
var l10nSV = new Intl.Collator("sv");  
console.log(l10nDE.compare("ä", "z") === -1); // true  
console.log(l10nSV.compare("ä", "z") === +1); // true
```

i. Sorting:

```
var list = [ "ä", "a", "z" ]; // In German, "ä" sorts with "a" Whereas in Swedish  
var l10nDE = new Intl.Collator("de");  
var l10nSV = new Intl.Collator("sv");  
console.log(list.sort(l10nDE.compare)) // [ "a", "ä", "z" ]  
console.log(list.sort(l10nSV.compare)) // [ "a", "z", "ä" ]
```

[↑ Back to Top](#)

323. What is for...of statement

The for...of statement creates a loop iterating over iterable objects or elements such as built-in String, Array, Array-like objects (like arguments or NodeList), TypedArray, Map, Set, and user-defined iterables. The basic usage of for...of statement on arrays would be as below,

```
let arrayIterable = [10, 20, 30, 40, 50];  
  
for (let value of arrayIterable) {  
    value ++;  
    console.log(value); // 11 21 31 41 51  
}
```

[↑ Back to Top](#)

324. What is the output of below spread operator array

```
[... 'John Resig']
```

The output of the array is ['J', 'o', 'h', 'n', ' ', 'R', 'e', 's', ' ', 'i', 'g'] **Explanation:** The string is an iterable type and the spread operator within an array maps every character of an iterable to one element. Hence, each character of a string becomes an element within an Array.

[↑ Back to Top](#)

325. Is PostMessage secure

Yes, postMessages can be considered very secure as long as the programmer/developer is careful about checking the origin and source of an arriving message. But if you try to send/receive a message without verifying its source will create cross-site scripting attacks.

 [Back to Top](#)

326. What are the problems with postmessage target origin as wildcard

The second argument of postMessage method specifies which origin is allowed to receive the message. If you use the wildcard "*" as an argument then any origin is allowed to receive the message. In this case, there is no way for the sender window to know if the target window is at the target origin when sending the message. If the target window has been navigated to another origin, the other origin would receive the data. Hence, this may lead to XSS vulnerabilities.

```
targetWindow.postMessage(message, '*');
```

 [Back to Top](#)

327. How do you avoid receiving postMessages from attackers

Since the listener listens for any message, an attacker can trick the application by sending a message from the attacker's origin, which gives an impression that the receiver received the message from the actual sender's window. You can avoid this issue by validating the origin of the message on the receiver's end using the "message.origin" attribute. For examples, let's check the sender's origin <http://www.some-sender.com> on receiver side www.some-receiver.com,

```
//Listener on http://www.some-receiver.com/
window.addEventListener("message", function(message){
    if(/^http://www\.some-sender\.com$/ .test(message.origin)){
        console.log('You received the data from valid sender', message.data);
    }
});
```

 [Back to Top](#)

328. Can I avoid using postMessages completely

You cannot avoid using postMessages completely(or 100%). Even though your application doesn't use postMessage considering the risks, a lot of third party scripts use postMessage to communicate with the third party service. So your application might be using postMessage without your knowledge.

 [Back to Top](#)

329. Is postMessages synchronous

The postMessages are synchronous in IE8 browser but they are asynchronous in IE9 and all other modern browsers (i.e, IE9+, Firefox, Chrome, Safari). Due to this asynchronous behaviour, we use a callback mechanism when the postMessage is returned.

 [Back to Top](#)

330. What paradigm is Javascript

JavaScript is a multi-paradigm language, supporting imperative/procedural programming, Object-Oriented Programming and functional programming. JavaScript supports Object-Oriented Programming with prototypical inheritance.

 [Back to Top](#)

331. What is the difference between internal and external javascript

Internal JavaScript: It is the source code within the script tag. **External JavaScript:** The source code is stored in an external file(stored with .js extension) and referred with in the tag.

 [Back to Top](#)

332. Is JavaScript faster than server side script

Yes, JavaScript is faster than server side script. Because JavaScript is a client-side script it does not require any web server's help for its computation or calculation. So JavaScript is always faster than any server-side script like ASP, PHP, etc.

 [Back to Top](#)

333. How do you get the status of a checkbox

You can apply the `checked` property on the selected checkbox in the DOM. If the value is `True` means the checkbox is checked otherwise it is unchecked. For example, the below HTML checkbox element can be access using javascript as below,

```
<input type="checkbox" name="checkboxname" value="Agree"> Agree the conditions<
```

```
console.log(document.getElementById('checkboxname').checked); // true or false
```

 [Back to Top](#)

334. What is the purpose of double tilde operator

The double tilde operator(`~~`) is known as double NOT bitwise operator. This operator is going to be a quicker substitute for `Math.floor()`.

 [Back to Top](#)

335. How do you convert character to ASCII code

You can use the `String.prototype.charCodeAt()` method to convert string characters to ASCII numbers. For example, let's find ASCII code for the first letter of 'ABC' string,

```
"ABC".charCodeAt(0) // returns 65
```

Whereas `String.fromCharCode()` method converts numbers to equal ASCII characters.

```
String.fromCharCode(65,66,67); // returns 'ABC'
```

 [Back to Top](#)

336. What is ArrayBuffer

An `ArrayBuffer` object is used to represent a generic, fixed-length raw binary data buffer. You can create it as below,

```
let buffer = new ArrayBuffer(16); // create a buffer of length 16
alert(buffer.byteLength); // 16
```

To manipulate an `ArrayBuffer`, we need to use a "view" object.

```
//Create a DataView referring to the buffer
let view = new DataView(buffer);
```

 [Back to Top](#)

337. What is the output of below string expression

```
console.log("Welcome to JS world"[0])
```

The output of the above expression is "W". **Explanation:** The bracket notation with specific index on a string returns the character at a specific location. Hence, it returns the character "W" of the string. Since this is not supported in IE7 and below versions, you may need to use the `.charAt()` method to get the desired result.

 [Back to Top](#)

338. What is the purpose of Error object

The Error constructor creates an error object and the instances of error objects are thrown when runtime errors occur. The Error object can also be used as a base object for user-defined exceptions. The syntax of error object would be as below,

```
new Error([message[, fileName[, lineNumber]]])
```

You can throw user defined exceptions or errors using Error object in try...catch block as below,

```
try {
  if(withdraw > balance)
    throw new Error("Oops! You don't have enough balance");
} catch (e) {
  console.log(e.name + ': ' + e.message);
}
```

 [Back to Top](#)

339. What is the purpose of EvalError object

The EvalError object indicates an error regarding the global eval() function. Even though this exception is not thrown by JavaScript anymore, the EvalError object remains for compatibility. The syntax of this expression would be as below,

```
new EvalError([message[, fileName[, lineNumber]]])
```

You can throw EvalError with in try...catch block as below,

```
try {
  throw new EvalError('Eval function error', 'someFile.js', 100);
} catch (e) {
  console.log(e.message, e.name, e.fileName);           // "Eval function error"
```

 [Back to Top](#)

340. What are the list of cases error thrown from non-strict mode to strict mode

When you apply 'use strict'; syntax, some of the below cases will throw a SyntaxError before executing the script

i. When you use Octal syntax

```
var n = 022;
```

i. Using with statement

ii. When you use delete operator on a variable name

iii. Using eval or arguments as variable or function argument name

iv. When you use newly reserved keywords

v. When you declare a function in a block

```
if (someCondition) { function f() {} }
```

Hence, the errors from above cases are helpful to avoid errors in development/production environments.

[↑ Back to Top](#)

341. Is all objects have prototypes

No. All objects have prototypes except for the base object which is created by the user, or an object that is created using the new keyword.

[↑ Back to Top](#)

342. What is the difference between a parameter and an argument

Parameter is the variable name of a function definition whereas an argument represents the value given to a function when it is invoked. Let's explain this with a simple function

```
function myFunction(parameter1, parameter2, parameter3) {
  console.log(arguments[0]) // "argument1"
  console.log(arguments[1]) // "argument2"
  console.log(arguments[2]) // "argument3"
}
myFunction("argument1", "argument2", "argument3")
```

[↑ Back to Top](#)

343. What is the purpose of some method in arrays

The some() method is used to test whether at least one element in the array passes the test implemented by the provided function. The method returns a boolean value. Let's take an example to test for any odd elements,

```
var array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

var odd = element => element % 2 !== 0;

console.log(array.some(odd)); // true (the odd element exists)
```

[↑ Back to Top](#)

344. How do you combine two or more arrays

The concat() method is used to join two or more arrays by returning a new array containing all the elements. The syntax would be as below,

```
array1.concat(array2, array3, ..., arrayX)
```

Let's take an example of array's concatenation with veggies and fruits arrays,

```
var veggies = ["Tomato", "Carrot", "Cabbage"];
var fruits = ["Apple", "Orange", "Pears"];
var veggiesAndFruits = veggies.concat(fruits);
console.log(veggiesAndFruits); // Tomato, Carrot, Cabbage, Apple, Orange, Pears
```

[↑ Back to Top](#)

345. What is the difference between Shallow and Deep copy

There are two ways to copy an object,

Shallow Copy: Shallow copy is a bitwise copy of an object. A new object is created that has an exact copy of the values in the original object. If any of the fields of the object are references to other objects, just the reference addresses are copied i.e., only the memory address is copied.

Example

```
var empDetails = {
  name: "John", age: 25, expertise: "Software Developer"
}
```

to create a duplicate

```
var empDetailsShallowCopy = empDetails //Shallow copying!
```

if we change some property value in the duplicate one like this:

```
empDetailsShallowCopy.name = "Johnson"
```

The above statement will also change the name of `empDetails`, since we have a shallow copy. That means we're losing the original data as well.

Deep copy: A deep copy copies all fields, and makes copies of dynamically allocated memory pointed to by the fields. A deep copy occurs when an object is copied along with the objects to which it refers.

Example

```
var empDetails = {  
    name: "John", age: 25, expertise: "Software Developer"  
}
```

Create a deep copy by using the properties from the original object into new variable

```
var empDetailsDeepCopy = {  
    name: empDetails.name,  
    age: empDetails.age,  
    expertise: empDetails.expertise  
}
```

Now if you change `empDetailsDeepCopy.name`, it will only affect `empDetailsDeepCopy` & not `empDetails`

[↑ Back to Top](#)

346. How do you create specific number of copies of a string

The `repeat()` method is used to construct and return a new string which contains the specified number of copies of the string on which it was called, concatenated together. Remember that this method has been added to the ECMAScript 2015 specification. Let's take an example of Hello string to repeat it 4 times,

```
'Hello'.repeat(4); // 'HelloHelloHelloHello'
```

347. How do you return all matching strings against a regular expression

The `matchAll()` method can be used to return an iterator of all results matching a string against a regular expression. For example, the below example returns an array of matching string results against a regular expression,

```
let regexp = /Hello(\d?)/g;
```

```
let greeting = 'Hello1Hello2Hello3';

let greetingList = [...greeting.matchAll(regexp)];

console.log(greetingList[0]); //Hello1
console.log(greetingList[1]); //Hello2
console.log(greetingList[2]); //Hello3
```

[↑ Back to Top](#)

348. How do you trim a string at the beginning or ending

The `trim` method of string prototype is used to trim on both sides of a string. But if you want to trim especially at the beginning or ending of the string then you can use `trimStart/trimLeft` and `trimEnd/trimRight` methods. Let's see an example of these methods on a greeting message,

```
var greeting = '    Hello, Goodmorning!    ';

console.log(greeting); // "    Hello, Goodmorning!    "
console.log(greeting.trimStart()); // "Hello, Goodmorning!    "
console.log(greeting.trimLeft()); // "Hello, Goodmorning!    "

console.log(greeting.trimEnd()); // "    Hello, Goodmorning!"
console.log(greeting.trimRight()); // "    Hello, Goodmorning!"
```

[↑ Back to Top](#)

349. What is the output of below console statement with unary operator

Let's take console statement with unary operator as given below,

```
console.log(+ 'Hello');
```

The output of the above console log statement returns NaN. Because the element is prefixed by the unary operator and the JavaScript interpreter will try to convert that element into a number type. Since the conversion fails, the value of the statement results in NaN value.

[↑ Back to Top](#)

350. Does javascript uses mixins

[↑ Back to Top](#)

351. What is a thunk function

A thunk is just a function which delays the evaluation of the value. It doesn't take any arguments but gives the value whenever you invoke the thunk. i.e, It is used not to execute now but it will be sometime in the future. Let's take a synchronous example,

```
const add = (x,y) => x + y;  
  
const thunk = () => add(2,3);  
  
thunk() // 5
```

[↑ Back to Top](#)

352. What are asynchronous thunks

The asynchronous thunks are useful to make network requests. Let's see an example of network requests,

```
function fetchData(fn){  
  fetch('https://jsonplaceholder.typicode.com/todos/1')  
    .then(response => response.json())  
    .then(json => fn(json))  
}  
  
const asyncThunk = function (){  
  return fetchData(function getData(data){  
    console.log(data)  
  })  
}  
  
asyncThunk()
```

The `getData` function won't be called immediately but it will be invoked only when the data is available from API endpoint. The `setTimeout` function is also used to make our code asynchronous. The best real time example is redux state management library which uses the asynchronous thunks to delay the actions to dispatch.

[↑ Back to Top](#)

353. What is the output of below function calls

Code snippet:

```
const circle = {  
  radius: 20,  
  diameter() {  
    return this.radius * 2;  
  },
```

```
    perimeter: () => 2 * Math.PI * this.radius  
};
```

```
console.log(circle.diameter()); console.log(circle.perimeter());
```

Output:

The output is 40 and NaN. Remember that diameter is a regular function, whereas the value of perimeter is an arrow function. The `this` keyword of a regular function(i.e, diameter) refers to the surrounding scope which is a class(i.e, Shape object). Whereas this keyword of perimeter function refers to the surrounding scope which is a window object. Since there is no radius property on window objects it returns an undefined value and the multiple of number value returns NaN value.

[↑ Back to Top](#)

354. How to remove all line breaks from a string

The easiest approach is using regular expressions to detect and replace newlines in the string. In this case, we use replace function along with string to replace with, which in our case is an empty string.

```
function remove_linebreaks( var message ) {  
    return message.replace( /[\\r\\n]+/gm, "" );  
}
```

In the above expression, g and m are for global and multiline flags.

[↑ Back to Top](#)

355. What is the difference between reflow and repaint

A *repaint* occurs when changes are made which affect the visibility of an element, but not its layout. Examples of this include outline, visibility, or background color. A *reflow* involves changes that affect the layout of a portion of the page (or the whole page). Resizing the browser window, changing the font, content changing (such as user typing text), using JavaScript methods involving computed styles, adding or removing elements from the DOM, and changing an element's classes are a few of the things that can trigger reflow. Reflow of an element causes the subsequent reflow of all child and ancestor elements as well as any elements following it in the DOM.

[↑ Back to Top](#)

356. What happens with negating an array

Negating an array with ! character will coerce the array into a boolean. Since Arrays are considered to be truthy So negating it will return false .

```
console.log(![]); // false
```

 [Back to Top](#)

357. What happens if we add two arrays

If you add two arrays together, it will convert them both to strings and concatenate them. For example, the result of adding arrays would be as below,

```
console.log(['a'] + ['b']); // "ab"
console.log([] + []); // ""
console.log(![] + []); // "false", because ![] returns false.
```

 [Back to Top](#)

358. What is the output of prepend additive operator on falsy values

If you prepend the additive(+) operator on falsy values(null, undefined, NaN, false, ""), the falsy value converts to a number value zero. Let's display them on browser console as below,

```
console.log(+null); // 0
console.log(+undefined); // NaN
console.log(+false); // 0
console.log(+NaN); // NaN
console.log(+ ""); // 0
```

 [Back to Top](#)

359. How do you create self string using special characters

The self string can be formed with the combination of []()!+ characters. You need to remember the below conventions to achieve this pattern.

- i. Since Arrays are truthful values, negating the arrays will produce false: ![] === false
- ii. As per JavaScript coercion rules, the addition of arrays together will toString them: [] + [] === ""
- iii. Prepend an array with + operator will convert an array to false, the negation will make it true and finally converting the result will produce value '1': +(!![]) === 1

By applying the above rules, we can derive below conditions

```
![] + [] === "false"  
+!+[] === 1
```

Now the character pattern would be created as below,

 Back to Top

360. How do you remove falsy values from an array

You can apply the filter method on the array by passing Boolean as a parameter. This way it removes all falsy values(0, undefined, null, false and "") from the array.

```
const myArray = [false, null, 1, 5, undefined]
myArray.filter(Boolean); // [1, 5] // is same as myArray.filter(x => x);
```

 Back to Top

361. How do you get unique values of an array

You can get unique values of an array with the combination of `Set` and rest expression/`spread(...)` syntax.

```
console.log([...new Set([1, 2, 4, 4, 3])]); // [1, 2, 4, 3]
```

 Back to Top

362. What is destructuring aliases

Sometimes you would like to have a destructured variable with a different name than the property name. In that case, you'll use a `: newName` to specify a name for the variable. This process is called destructuring aliases.

```
const obj = { x: 1 };
// Grabs obj.x as as { otherName }
const { x: otherName } = obj;
```

[↑ Back to Top](#)

363. How do you map the array values without using map method

You can map the array values without using the `map` method by just using the `from` method of Array. Let's map city names from Countries array,

```
const countries = [
  { name: 'India', capital: 'Delhi' },
  { name: 'US', capital: 'Washington' },
  { name: 'Russia', capital: 'Moscow' },
  { name: 'Singapore', capital: 'Singapore' },
  { name: 'China', capital: 'Beijing' },
  { name: 'France', capital: 'Paris' },
];

const cityNames = Array.from(countries, ({ capital }) => capital);
console.log(cityNames); // ['Delhi', 'Washington', 'Moscow', 'Singapore', 'Beijing'
```

[↑ Back to Top](#)

364. How do you empty an array

You can empty an array quickly by setting the array length to zero.

```
let cities = ['Singapore', 'Delhi', 'London'];
cities.length = 0; // cities becomes []
```

[↑ Back to Top](#)

365. How do you rounding numbers to certain decimals

You can round numbers to a certain number of decimals using `toFixed` method from native javascript.

```
let pie = 3.141592653;
pie = pie.toFixed(3); // 3.142
```

[↑ Back to Top](#)

366. What is the easiest way to convert an array to an object

You can convert an array to an object with the same data using spread(...) operator.

```
var fruits = ["banana", "apple", "orange", "watermelon"];
var fruitsObject = {...fruits};
console.log(fruitsObject); // {0: "banana", 1: "apple", 2: "orange", 3: "watermelc
```

[↑ Back to Top](#)

367. How do you create an array with some data

You can create an array with some data or an array with the same values using `fill` method.

```
var newArray = new Array(5).fill("0");
console.log(newArray); // ["0", "0", "0", "0", "0"]
```

[↑ Back to Top](#)

368. What are the placeholders from console object

Below are the list of placeholders available from console object,

- i. %o — It takes an object,
- ii. %s — It takes a string,
- iii. %d — It is used for a decimal or integer These placeholders can be represented in the `console.log` as below

```
const user = { "name": "John", "id": 1, "city": "Delhi"};
console.log("Hello %s, your details %o are available in the object form", "John",
```

[↑ Back to Top](#)

369. Is it possible to add CSS to console messages

Yes, you can apply CSS styles to console messages similar to html text on the web page.

```
console.log('%c The text has blue color, with large font and red background', 'color: blue; font-size: x-large; background-color: red');
```

The text will be displayed as below,

```
> console.log('%c Color of the text', 'color: blue; font-size: x-large; background-color: red');
```

Color of the text

vendors~main.51281d83.chunk.js:1

Note: All CSS styles can be applied to console messages.

[↑ Back to Top](#)

370. What is the purpose of dir method of console object

The `console.dir()` is used to display an interactive list of the properties of the specified JavaScript object as JSON.

```
const user = { "name": "John", "id": 1, "city": "Delhi"};
console.dir(user);
```

The user object displayed in JSON representation

```
> const user = { "name": "John", "id": 1, "city": "Delhi"};
  console.dir(user);

  ▼ Object ⓘ
    name: "John"
    id: 1
    city: "Delhi"
  ► __proto__: Object
```

[↑ Back to Top](#)

371. Is it possible to debug HTML elements in console

Yes, it is possible to get and debug HTML elements in the console just like inspecting elements.

```
const element = document.getElementsByTagName("body")[0];
console.log(element);
```

It prints the HTML element in the console,

```
> const element = document.getElementsByTagName("body")[0];
< undefined
> console.log(element);
  ► <body class="question-page unified-theme">...</body>
< undefined
> |
```

[↑ Back to Top](#)

372. How do you display data in a tabular format using console object

The `console.table()` is used to display data in the console in a tabular format to visualize complex arrays or objects.

```
const users = [{ "name": "John", "id": 1, "city": "Delhi"}, { "name": "Max", "id": 2, "city": "London"}, { "name": "Rod", "id": 3, "city": "Paris"}];
console.table(users);
```

The data visualized in a table format,

```
> const users = [{ "name": "John", "id": 1, "city": "Delhi"}, { "name": "Max", "id": 2, "city": "London"}, { "name": "Rod", "id": 3, "city": "Paris"}];
< undefined
> console.table(users);
VM92:1


| (index) | name   | id | city     |
|---------|--------|----|----------|
| 0       | "John" | 1  | "Delhi"  |
| 1       | "Max"  | 2  | "London" |
| 2       | "Rod"  | 3  | "Paris"  |


▶ Array(3)
```

Not: Remember that `console.table()` is not supported in IE.

[↑ Back to Top](#)

373. How do you verify that an argument is a Number or not

The combination of `isNaN` and `isFinite` methods are used to confirm whether an argument is a number or not.

```
function isNumber(n){
    return !isNaN(parseFloat(n)) && isFinite(n);
}
```

[↑ Back to Top](#)

374. How do you create copy to clipboard button

You need to select the content(using `.select()` method) of the input element and execute the copy command with `execCommand` (i.e, `execCommand('copy')`). You can also execute other system commands like cut and paste.

```
document.querySelector("#copy-button").onclick = function() {
    // Select the content
    document.querySelector("#copy-input").select();
    // Copy to the clipboard
    document.execCommand('copy');
};
```

[↑ Back to Top](#)

375. What is the shortcut to get timestamp

You can use `new Date().getTime()` to get the current timestamp. There is an alternative shortcut to get the value.

```
console.log(+new Date());
console.log(Date.now());
```

[↑ Back to Top](#)

376. How do you flattening multi dimensional arrays

Flattening bi-dimensional arrays is trivial with Spread operator.

```
const biDimensionalArr = [11, [22, 33], [44, 55], [66, 77], 88, 99];
const flattenArr = [...biDimensionalArr]; // [11, 22, 33, 44, 55, 66, 77,
```

But you can make it work with multi-dimensional arrays by recursive calls,

```
function flattenMultiArray(arr) {
  const flattened = [...].concat(...arr);
  return flattened.some(item => Array.isArray(item)) ? flattenMultiArray(flattened)
}
const multiDimensionalArr = [11, [22, 33], [44, [55, 66, [77, [88]]], 99]];
const flatArr = flattenMultiArray(multiDimensionalArr); // [11, 22, 33, 44, 55, 66, 77, 88]
```

[↑ Back to Top](#)

377. What is the easiest multi condition checking

You can use `indexof` to compare input with multiple values instead of checking each value as one condition.

```
// Verbose approach
if (input === 'first' || input === 1 || input === 'second' || input === 2) {
  someFunction();
}
// Shortcut
if (['first', 1, 'second', 2].indexOf(input) !== -1) {
  someFunction();
}
```

[↑ Back to Top](#)

378. How do you capture browser back button

The `window.onbeforeunload` method is used to capture browser back button events. This is helpful to warn users about losing the current data.

```
window.onbeforeunload = function() {
    alert("Your work will be lost");
};
```

 [Back to Top](#)

379. How do you disable right click in the web page

The right click on the page can be disabled by returning false from the `oncontextmenu` attribute on the body element.

```
<body oncontextmenu="return false;">
```

 [Back to Top](#)

380. What are wrapper objects

Primitive Values like string, number and boolean don't have properties and methods but they are temporarily converted or coerced to an object(Wrapper object) when you try to perform actions on them. For example, if you apply `toUpperCase()` method on a primitive string value, it does not throw an error but returns uppercase of the string.

```
let name = "john";

console.log(name.toUpperCase()); // Behind the scenes treated as console.log(new
```

i.e, Every primitive except null and undefined have Wrapper Objects and the list of wrapper objects are String, Number, Boolean, Symbol and BigInt.

 [Back to Top](#)

381. What is AJAX

AJAX stands for Asynchronous JavaScript and XML and it is a group of related technologies(HTML, CSS, JavaScript, XMLHttpRequest API etc) used to display data asynchronously. i.e. We can send data to the server and get data from the server without reloading the web page.

 [Back to Top](#)

382. What are the different ways to deal with Asynchronous Code

Below are the list of different ways to deal with Asynchronous code.

- i. Callbacks
- ii. Promises
- iii. Async/await
- iv. Third-party libraries such as `async.js`, `bluebird` etc

 [Back to Top](#)

383. How to cancel a fetch request

Until a few days back, One shortcoming of native promises is no direct way to cancel a fetch request. But the new `AbortController` from js specification allows you to use a signal to abort one or multiple fetch calls. The basic flow of cancelling a fetch request would be as below,

- i. Create an `AbortController` instance
 - ii. Get the `signal` property of an instance and pass the signal as a fetch option for `signal`
 - iii. Call the `AbortController`'s `abort` property to cancel all fetches that use that signal
- For example, let's pass the same signal to multiple fetch calls will cancel all requests with that signal,

```
const controller = new AbortController();
const { signal } = controller;

fetch("http://localhost:8000", { signal }).then(response => {
    console.log(`Request 1 is complete!`);
}).catch(e => {
    if(e.name === "AbortError") {
        // We know it's been canceled!
    }
});

fetch("http://localhost:8000", { signal }).then(response => {
    console.log(`Request 2 is complete!`);
}).catch(e => {
    if(e.name === "AbortError") {
        // We know it's been canceled!
    }
});

// Wait 2 seconds to abort both requests
setTimeout(() => controller.abort(), 2000);
```

 [Back to Top](#)

384. What is web speech API

Web speech API is used to enable modern browsers recognize and synthesize speech(i.e, voice data into web apps). This API has been introduced by W3C Community in the year 2012. It has two main parts,

- i. **SpeechRecognition (Asynchronous Speech Recognition or Speech-to-Text):** It provides the ability to recognize voice context from an audio input and respond accordingly. This is accessed by the `SpeechRecognition` interface. The below example shows on how to use this API to get text from speech,

```
window.SpeechRecognition = window.webkitSpeechRecognition || window.SpeechRecognition
const recognition = new window.SpeechRecognition();
recognition.onresult = (event) => { // SpeechRecognitionEvent type
  const speechToText = event.results[0][0].transcript;
  console.log(speechToText);
}
recognition.start();
```

In this API, browser is going to ask you for permission to use your microphone

- i. **SpeechSynthesis (Text-to-Speech):** It provides the ability to recognize voice context from an audio input and respond. This is accessed by the `SpeechSynthesis` interface. For example, the below code is used to get voice/speech from text,

```
if('speechSynthesis' in window){
  var speech = new SpeechSynthesisUtterance('Hello World!');
  speech.lang = 'en-US';
  window.speechSynthesis.speak(speech);
}
```

The above examples can be tested on chrome(33+) browser's developer console. **Note:** This API is still a working draft and only available in Chrome and Firefox browsers(ofcourse Chrome only implemented the specification) [↑ Back to Top](#)

385. What is minimum timeout throttling

Both browser and NodeJS javascript environments throttle with a minimum delay that is greater than 0ms. That means even though setting a delay of 0ms will not happen instantaneously. **Browsers:** They have a minimum delay of 4ms. This throttle occurs when successive calls are triggered due to callback nesting(certain depth) or after a certain number of successive intervals. Note: The older browsers have a minimum delay of 10ms. **Nodejs:** They have a minimum delay of 1ms. This throttle happens when the delay is larger than 2147483647 or less than 1. The best example to explain this timeout throttling behavior is the order of below code snippet.

```
function runMeFirst() {
  console.log('My script is initialized');
}
```

```
setTimeout(runMeFirst, 0);
console.log('Script loaded');
```

and the output would be in

```
Script loaded
My script is initialized
```

If you don't use `setTimeout`, the order of logs will be sequential.

```
function runMeFirst() {
  console.log('My script is initialized');
}
runMeFirst();
console.log('Script loaded');
```

and the output is,

```
My script is initialized
Script loaded
```

[↑ Back to Top](#)

386. How do you implement zero timeout in modern browsers

You can't use `setTimeout(fn, 0)` to execute the code immediately due to minimum delay of greater than 0ms. But you can use `window.postMessage()` to achieve this behavior.

[↑ Back to Top](#)

387. What are tasks in event loop

A task is any javascript code/program which is scheduled to be run by the standard mechanisms such as initially starting to run a program, run an event callback, or an interval or timeout being fired. All these tasks are scheduled on a task queue. Below are the list of use cases to add tasks to the task queue,

- i. When a new javascript program is executed directly from console or running by the `<script>` element, the task will be added to the task queue.
- ii. When an event fires, the event callback added to task queue
- iii. When a `setTimeout` or `setInterval` is reached, the corresponding callback added to task queue

[↑ Back to Top](#)

388. What is microtask

Microtask is the javascript code which needs to be executed immediately after the currently executing task/microtask is completed. They are kind of blocking in nature. i.e, The main thread will be blocked until the microtask queue is empty. The main sources of microtasks are `Promise.resolve`, `Promise.reject`, `MutationObservers`, `IntersectionObservers` etc

Note: All of these microtasks are processed in the same turn of the event loop. [↑ Back to Top](#)

389. What are different event loops

[↑ Back to Top](#)

390. What is the purpose of queueMicrotask

[↑ Back to Top](#)

391. How do you use javascript libraries in typescript file

It is known that not all JavaScript libraries or frameworks have TypeScript declaration files. But if you still want to use libraries or frameworks in our TypeScript files without getting compilation errors, the only solution is `declare` keyword along with a variable declaration. For example, let's imagine you have a library called `customLibrary` that doesn't have a TypeScript declaration and have a namespace called `customLibrary` in the global namespace. You can use this library in typescript code as below,

```
declare var customLibrary;
```

In the runtime, typescript will provide the type to the `customLibrary` variable as `any` type. The another alternative without using `declare` keyword is below

```
var customLibrary: any;
```

[↑ Back to Top](#)

392. What are the differences between promises and observables

Some of the major difference in a tabular form

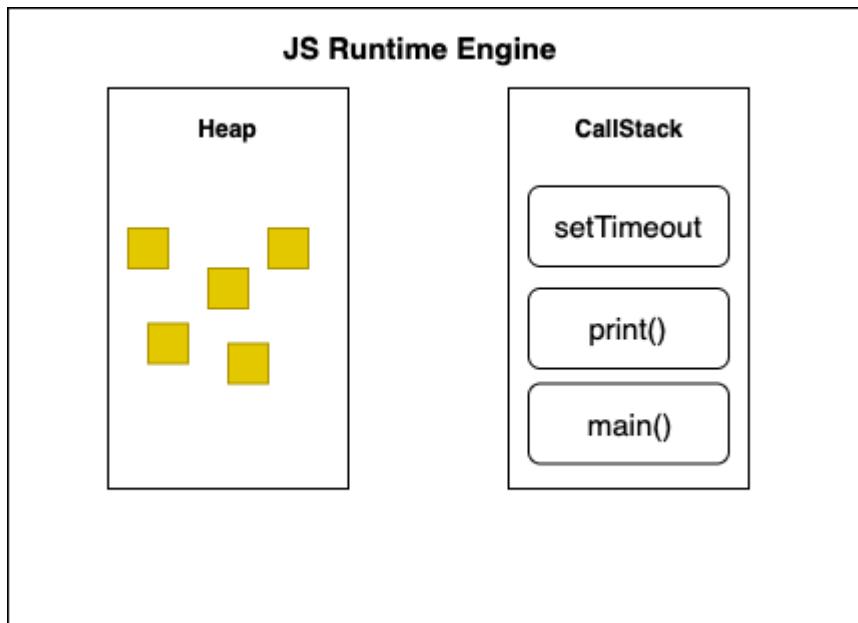
| Promises | Observables |
|-------------------------------------|--|
| Emits only a single value at a time | Emits multiple values over a period of time(stream of values ranging from 0 to |

| Promises | Observables |
|--|---|
| | multiple) |
| Eager in nature; they are going to be called immediately | Lazy in nature; they require subscription to be invoked |
| Promise is always asynchronous even though it resolved immediately | Observable can be either synchronous or asynchronous |
| Doesn't provide any operators | Provides operators such as map, forEach, filter, reduce, retry, and retryWhen etc |
| Cannot be canceled | Canceled by using unsubscribe() method |

[↑ Back to Top](#)

393. What is heap

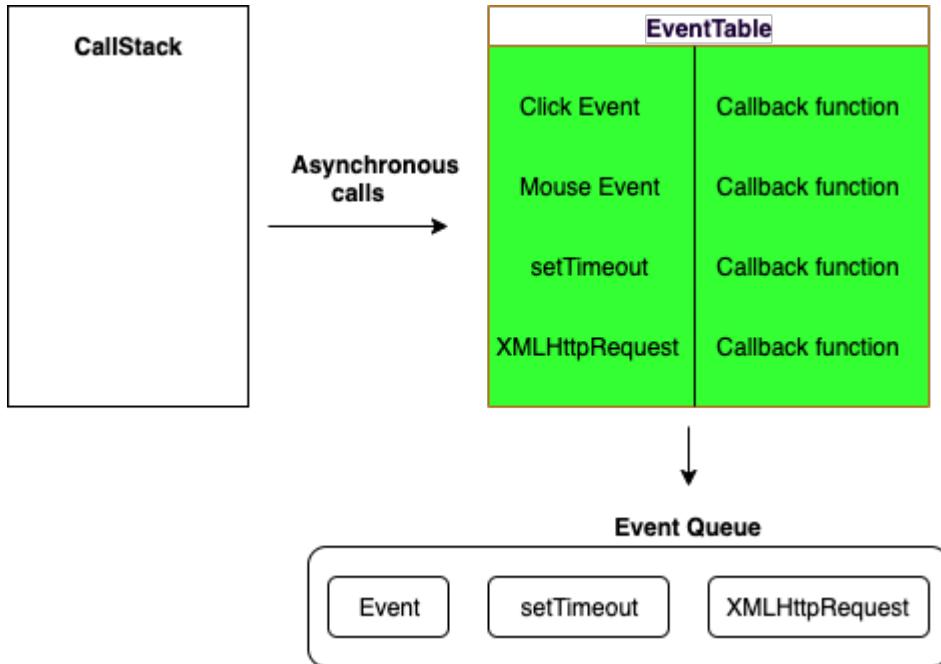
Heap(Or memory heap) is the memory location where objects are stored when we define variables. i.e, This is the place where all the memory allocations and de-allocation take place. Both heap and call-stack are two containers of JS runtime. Whenever runtime comes across variables and function declarations in the code it stores them in the Heap.



[↑ Back to Top](#)

394. What is an event table

Event Table is a data structure that stores and keeps track of all the events which will be executed asynchronously like after some time interval or after the resolution of some API requests. i.e Whenever you call a setTimeout function or invoke async operation, it is added to the Event Table. It doesn't execute functions on its own. The main purpose of the event table is to keep track of events and send them to the Event Queue as shown in the below diagram.



[↑ Back to Top](#)

395. What is a microTask queue

Microtask Queue is the new queue where all the tasks initiated by promise objects get processed before the callback queue. The microtasks queue are processed before the next rendering and painting jobs. But if these microtasks are running for a long time then it leads to visual degradation.

[↑ Back to Top](#)

396. What is the difference between shim and polyfill

A shim is a library that brings a new API to an older environment, using only the means of that environment. It isn't necessarily restricted to a web application. For example, es5-shim.js is used to emulate ES5 features on older browsers (mainly pre IE9). Whereas polyfill is a piece of code (or plugin) that provides the technology that you, the developer, expect the browser to provide natively. In a simple sentence, A polyfill is a shim for a browser API.

[↑ Back to Top](#)

397. How do you detect primitive or non primitive value type

In JavaScript, primitive types include boolean, string, number, BigInt, null, Symbol and undefined. Whereas non-primitive types include the Objects. But you can easily identify them with the below function,

```
var myPrimitive = 30;
var myNonPrimitive = {};
function isPrimitive(val) {
    return Object(val) !== val;
}

isPrimitive(myPrimitive);
isPrimitive(myNonPrimitive);
```

If the value is a primitive data type, the Object constructor creates a new wrapper object for the value. But If the value is a non-primitive data type (an object), the Object constructor will give the same object.

 [Back to Top](#)

398. What is babel

Babel is a JavaScript transpiler to convert ECMAScript 2015+ code into a backwards compatible version of JavaScript in current and older browsers or environments. Some of the main features are listed below,

- i. Transform syntax
- ii. Polyfill features that are missing in your target environment (using @babel/polyfill)
- iii. Source code transformations (or codemods)

 [Back to Top](#)

399. Is Node.js completely single threaded

Node is a single thread, but some of the functions included in the Node.js standard library(e.g, fs module functions) are not single threaded. i.e, Their logic runs outside of the Node.js single thread to improve the speed and performance of a program.

 [Back to Top](#)

400. What are the common use cases of observables

Some of the most common use cases of observables are web sockets with push notifications, user input changes, repeating intervals, etc

 [Back to Top](#)

401. What is RxJS

RxJS (Reactive Extensions for JavaScript) is a library for implementing reactive programming using observables that makes it easier to compose asynchronous or callback-based code. It also provides utility functions for creating and working with observables.

[↑ Back to Top](#)

402. What is the difference between Function constructor and function declaration

The functions which are created with `Function constructor` do not create closures to their creation contexts but they are always created in the global scope. i.e, the function can access its own local variables and global scope variables only. Whereas function declarations can access outer function variables(closures) too.

Let's see this difference with an example,

Function Constructor:

```
var a = 100;
function createFunction() {
    var a = 200;
    return new Function('return a;');
}
console.log(createFunction()); // 100
```

Function declaration:

```
var a = 100;
function createFunction() {
    var a = 200;
    return function func() {
        return a;
    }
}
console.log(createFunction()); // 200
```

[↑ Back to Top](#)

403. What is a Short circuit condition

Short circuit conditions are meant for condensed way of writing simple if statements. Let's demonstrate the scenario using an example. If you would like to login to a portal with an authentication condition, the expression would be as below,

```
if (authenticate) {
    loginToPorta();
```

```
}
```

Since the javascript logical operators evaluated from left to right, the above expression can be simplified using `&&` logical operator

```
authenticate && loginToPorta();
```

 [Back to Top](#)

404. What is the easiest way to resize an array

The `length` property of an array is useful to resize or empty an array quickly. Let's apply `length` property on number array to resize the number of elements from 5 to 2,

```
var array = [1, 2, 3, 4, 5];
console.log(array.length); // 5

array.length = 2;
console.log(array.length); // 2
console.log(array); // [1,2]
```

and the array can be emptied too

```
var array = [1, 2, 3, 4, 5];
array.length = 0;
console.log(array.length); // 0
console.log(array); // []
```

 [Back to Top](#)

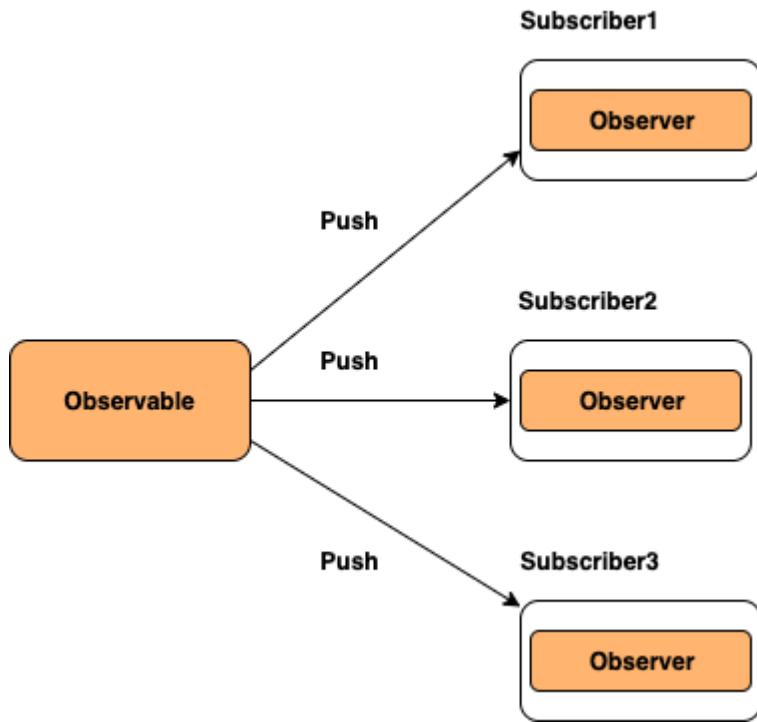
405. What is an observable

An Observable is basically a function that can return a stream of values either synchronously or asynchronously to an observer over time. The consumer can get the value by calling `subscribe()` method. Let's look at a simple example of an Observable

```
import { Observable } from 'rxjs';

const observable = new Observable(observer => {
  setTimeout(() => {
    observer.next('Message from a Observable!');
  }, 3000);
});

observable.subscribe(value => console.log(value));
```



Note: Observables are not part of the JavaScript language yet but they are being proposed to be added to the language

[↑ Back to Top](#)

406. What is the difference between function and class declarations

The main difference between function declarations and class declarations is `hoisting`. The function declarations are hoisted but not class declarations.

Classes:

```

const user = new User(); // ReferenceError

class User {}
  
```

Constructor Function:

```

const user = new User(); // No error

function User() {
}
  
```

[↑ Back to Top](#)

407. What is an async function

An `async` function is a function declared with the `async` keyword which enables asynchronous, promise-based behavior to be written in a cleaner style by avoiding promise chains. These functions can contain zero or more `await` expressions.

Let's take a below `async` function example,

```
async function logger() {  
  
  let data = await fetch('http://someapi.com/users'); // pause until fetch returns  
  console.log(data)  
}  
logger();
```

It is basically syntax sugar over ES2015 promises and generators.

 [Back to Top](#)

408. How do you prevent promises swallowing errors

While using asynchronous code, JavaScript's ES6 promises can make your life a lot easier without having callback pyramids and error handling on every second line. But Promises have some pitfalls and the biggest one is swallowing errors by default.

Let's say you expect to print an error to the console for all the below cases,

```
Promise.resolve('promised value').then(function() {  
  throw new Error('error');  
});  
  
Promise.reject('error value').catch(function() {  
  throw new Error('error');  
});  
  
new Promise(function(resolve, reject) {  
  throw new Error('error');  
});
```

But there are many modern JavaScript environments that won't print any errors. You can fix this problem in different ways,

- i. **Add catch block at the end of each chain:** You can add catch block to the end of each of your promise chains

```
Promise.resolve('promised value').then(function() {
    throw new Error('error');
}).catch(function(error) {
    console.error(error.stack);
});
```

But it is quite difficult to type for each promise chain and verbose too.

- ii. **Add done method:** You can replace first solution's then and catch blocks with done method

```
Promise.resolve('promised value').done(function() {
    throw new Error('error');
});
```

Let's say you want to fetch data using HTTP and later perform processing on the resulting data asynchronously. You can write done block as below,

```
getDataFromHttp()
  .then(function(result) {
    return processDataAsync(result);
})
.done(function(processed) {
  displayData(processed);
});
```

In future, if the processing library API changed to synchronous then you can remove done block as below,

```
getDataFromHttp()
  .then(function(result) {
    return displayData(processDataAsync(result));
})
```

and then you forgot to add done block to then block leads to silent errors.

- iii. **Extend ES6 Promises by Bluebird:** Bluebird extends the ES6 Promises API to avoid the issue in the second solution. This library has a "default" onRejection handler which will print all errors from rejected Promises to stderr. After installation, you can process unhandled rejections

```
Promise.onPossiblyUnhandledRejection(function(error){
    throw error;
});
```

and discard a rejection, just handle it with an empty catch

```
Promise.reject('error value').catch(function() {});
```

[↑ Back to Top](#)

409. What is deno

Deno is a simple, modern and secure runtime for JavaScript and TypeScript that uses V8 JavaScript engine and the Rust programming language.

[↑ Back to Top](#)

410. How do you make an object iterable in javascript

By default, plain objects are not iterable. But you can make the object iterable by defining a `Symbol.iterator` property on it.

Let's demonstrate this with an example,

```
const collection = {
  one: 1,
  two: 2,
  three: 3,
  [Symbol.iterator]() {
    const values = Object.keys(this);
    let i = 0;
    return {
      next: () => {
        return {
          value: this[values[i++]],
          done: i > values.length
        }
      }
    };
  }
};

const iterator = collection[Symbol.iterator]();

console.log(iterator.next()); // → {value: 1, done: false}
console.log(iterator.next()); // → {value: 2, done: false}
console.log(iterator.next()); // → {value: 3, done: false}
console.log(iterator.next()); // → {value: undefined, done: true}
```

The above process can be simplified using a generator function,

```
const collection = {
```

```

one: 1,
two: 2,
three: 3,
[Symbol.iterator]: function * () {
  for (let key in this) {
    yield this[key];
  }
};
const iterator = collection[Symbol.iterator]();
console.log(iterator.next()); // {value: 1, done: false}
console.log(iterator.next()); // {value: 2, done: false}
console.log(iterator.next()); // {value: 3, done: false}
console.log(iterator.next()); // {value: undefined, done: true}

```

[↑ Back to Top](#)

411. What is a Proper Tail Call

First, we should know about tail call before talking about "Proper Tail Call". A tail call is a subroutine or function call performed as the final action of a calling function. Whereas **Proper tail call(PTC)** is a technique where the program or code will not create additional stack frames for a recursion when the function call is a tail call.

For example, the below classic or head recursion of factorial function relies on stack for each step. Each step need to be processed upto `n * factorial(n - 1)`

```

function factorial(n) {
  if (n === 0) {
    return 1
  }
  return n * factorial(n - 1)
}
console.log(factorial(5)); //120

```

But if you use Tail recursion functions, they keep passing all the necessary data it needs down the recursion without relying on the stack.

```

function factorial(n, acc = 1) {
  if (n === 0) {
    return acc
  }
  return factorial(n - 1, n * acc)
}
console.log(factorial(5)); //120

```

The above pattern returns the same output as the first one. But the accumulator keeps track of total as an argument without using stack memory on recursive calls.

[↑ Back to Top](#)

412. How do you check an object is a promise or not

If you don't know if a value is a promise or not, wrapping the value as `Promise.resolve(value)` which returns a promise

```
function isPromise(object){  
    if(Promise && Promise.resolve){  
        return Promise.resolve(object) == object;  
    }else{  
        throw "Promise not supported in your environment"  
    }  
}  
  
var i = 1;  
var promise = new Promise(function(resolve,reject){  
    resolve()  
});  
  
console.log(isPromise(i)); // false  
console.log(isPromise(p)); // true
```

Another way is to check for `.then()` handler type

```
function isPromise(value) {  
    return Boolean(value && typeof value.then === 'function');  
}  
var i = 1;  
var promise = new Promise(function(resolve,reject){  
    resolve()  
});  
  
console.log(isPromise(i)) // false  
console.log(isPromise(promise)); // true
```

[↑ Back to Top](#)

413. How to detect if a function is called as constructor

You can use `new.target` pseudo-property to detect whether a function was called as a constructor(using the `new` operator) or as a regular function call.

- i. If a constructor or function invoked using the `new` operator, `new.target` returns a reference to the constructor or function.
- ii. For function calls, `new.target` is `undefined`.

```
function MyFunc() {
```

```

    if (new.target) {
        console.log('called with new');
    } else {
        console.log('not called with new');
    }
}

new Myfunc(); // called with new
Myfunc(); // not called with new
Myfunc.call({}); not called with new

```

[↑ Back to Top](#)

414. What are the differences between arguments object and rest parameter

There are three main differences between arguments object and rest parameters

- i. The arguments object is an array-like but not an array. Whereas the rest parameters are array instances.
- ii. The arguments object does not support methods such as sort, map, forEach, or pop. Whereas these methods can be used in rest parameters.
- iii. The rest parameters are only the ones that haven't been given a separate name, while the arguments object contains all arguments passed to the function

[↑ Back to Top](#)

415. What are the differences between spread operator and rest parameter

Rest parameter collects all remaining elements into an array. Whereas Spread operator allows iterables(arrays / objects / strings) to be expanded into single arguments/elements. i.e, Rest parameter is opposite to the spread operator.

[↑ Back to Top](#)

416. What are the different kinds of generators

There are five kinds of generators,

- i. Generator function declaration:

```
function* myGenFunc() {
    yield 1;
    yield 2;
    yield 3;
}
const genObj = myGenFunc();
```

ii. Generator function expressions:

```
const myGenFunc = function* () {
    yield 1;
    yield 2;
    yield 3;
};
const genObj = myGenFunc();
```

iii. Generator method definitions in object literals:

```
const myObj = {
    * myGeneratorMethod() {
        yield 1;
        yield 2;
        yield 3;
    }
};
const genObj = myObj.myGeneratorMethod();
```

iv. Generator method definitions in class:

```
class MyClass {
    * myGeneratorMethod() {
        yield 1;
        yield 2;
        yield 3;
    }
}
const myObject = new MyClass();
const genObj = myObject.myGeneratorMethod();
```

v. Generator as a computed property:

```
const SomeObj = {
    *[Symbol.iterator] () {
        yield 1;
        yield 2;
        yield 3;
    }
}
```

```
}

console.log(Array.from(SomeObj)); // [ 1, 2, 3 ]
```

[↑ Back to Top](#)

417. What are the built-in iterables

Below are the list of built-in iterables in javascript,

- i. Arrays and TypedArrays
- ii. Strings: Iterate over each character or Unicode code-points
- iii. Maps: iterate over its key-value pairs
- iv. Sets: iterates over their elements
- v. arguments: An array-like special variable in functions
- vi. DOM collection such as NodeList

[↑ Back to Top](#)

418. What are the differences between for...of and for...in statements

Both for...in and for...of statements iterate over js data structures. The only difference is over what they iterate:

- i. for..in iterates over all enumerable property keys of an object
- ii. for..of iterates over the values of an iterable object.

Let's explain this difference with an example,

```
let arr = ['a', 'b', 'c'];

arr.newProp = 'newValue';

// key are the property keys
for (let key in arr) {
  console.log(key);
}

// value are the property values
for (let value of arr) {
  console.log(value);
}
```

Since for..in loop iterates over the keys of the object, the first loop logs 0, 1, 2 and newProp while iterating over the array object. The for..of loop iterates over the values of a arr data structure and logs a, b, c in the console.

 [Back to Top](#)

419. How do you define instance and non-instance properties

The Instance properties must be defined inside of class methods. For example, name and age properties defined inside constructor as below,

```
class Person {  
    constructor(name, age) {  
        this.name = name;  
        this.age = age;  
    }  
}
```

But Static(class) and prototype data properties must be defined outside of the ClassBody declaration. Let's assign the age value for Person class as below,

```
Person.staticAge = 30;  
Person.prototype.prototypeAge = 40;
```

 [Back to Top](#)

420. What is the difference between isNaN and Number.isNaN?

- i. **isNaN**: The global function `isNaN` converts the argument to a Number and returns true if the resulting value is NaN.
- ii. **Number.isNaN**: This method does not convert the argument. But it returns true when the type is a Number and value is NaN.

Let's see the difference with an example,

```
isNaN('hello'); // true  
Number.isNaN('hello'); // false
```

 [Back to Top](#)

421. How to invoke an IIFE without any extra brackets?

Immediately Invoked Function Expressions(IIFE) requires a pair of parenthesis to wrap the function which contains set of statements.

```
(function(dt) {  
    console.log(dt.toLocaleTimeString());  
})(new Date());
```

Since both IIFE and void operator discard the result of an expression, you can avoid the extra brackets using void operator for IIFE as below,

```
void function(dt) {  
    console.log(dt.toLocaleTimeString());  
}(new Date());
```

[↑ Back to Top](#table-of-contents)

422. Is that possible to use expressions in switch cases?

You might have seen expressions used in switch condition but it is also possible to use for switch cases by assigning true value for the switch condition. Let's see the weather condition based on temparature as an example,

```
const weather = function getWeather(temp) {  
    switch(true) {  
        case temp < 0: return 'freezing';  
        case temp < 10: return 'cold';  
        case temp < 24: return 'cool';  
        default: return 'unknown';  
    }  
}(10);
```

[↑ Back to Top](#table-of-contents)

423. What is the easiest way to ignore promise errors?

The easiest and safest way to ignore promise errors is void that error. This approach is ESLint friendly too.

```
await promise.catch(e => void e);
```

 [Back to Top](#)

424. How do style the console output using CSS?

You can add CSS styling to the console output using the CSS format content specifier %c. The console string message can be appended after the specifier and CSS style in another argument. Let's print the red the color text using console.log and CSS specifier as below,

```
console.log("%cThis is a red text", "color:red");
```

It is also possible to add more styles for the content. For example, the font-size can be modified for the above text

```
console.log("%cThis is a red text with bigger font", "color:red; font-size:20px");
```

 [Back to Top](#)

Coding Exercise

1. What is the output of below code

```
var car = new Vehicle("Honda", "white", "2010", "UK");
console.log(car);

function Vehicle(model, color, year, country) {
    this.model = model;
    this.color = color;
    this.year = year;
    this.country = country;
}
```

- 1: Undefined
- 2: ReferenceError
- 3: null
- 4: {model: "Honda", color: "white", year: "2010", country: "UK"}

► Answer

 [Back to Top](#)

2. What is the output of below code

```
function foo() {
    let x = y = 0;
    x++;
    y++;
    return x;
}

console.log(foo(), typeof x, typeof y);
```

- 1: 1, undefined and undefined
- 2: ReferenceError: X is not defined
- 3: 1, undefined and number
- 4: 1, number and number

► Answer

 [Back to Top](#)

3. What is the output of below code

```
function main(){
    console.log('A');
    setTimeout(
        function print(){ console.log('B'); }
    ,0);
    console.log('C');
}
main();
```

- 1: A, B and C
- 2: B, A and C
- 3: A and C
- 4: A, C and B

► Answer

 [Back to Top](#)

4. What is the output of below equality check

```
console.log(0.1 + 0.2 === 0.3);
```

- 1: false
- 2: true

► Answer

 [Back to Top](#)

5. What is the output of below code

```
var y = 1;
if (function f(){}){
    y += typeof f;
}
console.log(y);
```

- 1: 1function
- 2: 1object
- 3: ReferenceError
- 4: 1undefined

► Answer

 [Back to Top](#)

6. What is the output of below code

```
function foo() {
    return
{
    message: "Hello World"
};
}
console.log(foo());
```

- 1: Hello World
- 2: Object {message: "Hello World"}
- 3: Undefined
- 4: SyntaxError

► Answer

 [Back to Top](#)

7. What is the output of below code

```
var myChars = ['a', 'b', 'c', 'd']
delete myChars[0];
console.log(myChars);
console.log(myChars[0]);
console.log(myChars.length);
```

- 1: [empty, 'b', 'c', 'd'], empty, 3
- 2: [null, 'b', 'c', 'd'], empty, 3
- 3: [empty, 'b', 'c', 'd'], undefined, 4
- 4: [null, 'b', 'c', 'd'], undefined, 4

► Answer

 [Back to Top](#)

8. What is the output of below code in latest Chrome

```
var array1 = new Array(3);
console.log(array1);

var array2 = [];
array2[2] = 100;
console.log(array2);

var array3 = [,,,];
console.log(array3);
```

- 1: [undefined × 3], [undefined × 2, 100], [undefined × 3]
- 2: [empty × 3], [empty × 2, 100], [empty × 3]
- 3: [null × 3], [null × 2, 100], [null × 3]
- 4: [], [100], []

► Answer

 [Back to Top](#)

9. What is the output of below code

```
const obj = {
  prop1: function() { return 0 },
  prop2() { return 1 },
  ['prop' + 3]() { return 2 }
}

console.log(obj.prop1());
console.log(obj.prop2());
console.log(obj.prop3());
```

- 1: 0, 1, 2

- 2: 0, { return 1 }, 2
- 3: 0, { return 1 }, { return 2 }
- 4: 0, 1, undefined

► Answer

 [Back to Top](#)

10. What is the output of below code

```
console.log(1 < 2 < 3);
console.log(3 > 2 > 1);
```

- 1: true, true
- 2: true, false
- 3: SyntaxError, SyntaxError,
- 4: false, false

► Answer

 [Back to Top](#)

11. What is the output of below code in non-strict mode

```
function printNumbers(first, second, first) {
  console.log(first, second, first);
}
printNumbers(1, 2, 3);
```

- 1: 1, 2, 3
- 2: 3, 2, 3
- 3: SyntaxError: Duplicate parameter name not allowed in this context
- 4: 1, 2, 1

► Answer

 [Back to Top](#)

12. What is the output of below code

```
const printNumbersArrow = (first, second, first) => {
  console.log(first, second, first);
}
printNumbersArrow(1, 2, 3);
```

- 1: 1, 2, 3
- 2: 3, 2, 3
- 3: SyntaxError: Duplicate parameter name not allowed in this context
- 4: 1, 2, 1

► Answer

 [Back to Top](#)

13. What is the output of below code

```
const arrowFunc = () => arguments.length;
console.log(arrowFunc(1, 2, 3));
```

- 1: ReferenceError: arguments is not defined
- 2: 3
- 3: undefined
- 4: null

► Answer

 [Back to Top](#)

14. What is the output of below code

```
console.log( String.prototype.trimLeft.name === 'trimLeft' );
console.log( String.prototype.trimLeft.name === 'trimStart' );
```

- 1: True, False
- 2: False, True

► Answer

 [Back to Top](#)

15. What is the output of below code

```
console.log(Math.max());
```

- 1: undefined
- 2: Infinity
- 3: 0
- 4: -Infinity

► Answer

 [Back to Top](#)

16. What is the output of below code

```
console.log(10 == [10]);  
console.log(10 == [[[[[[10]]]]]]);
```

- 1: True, True
- 2: True, False
- 3: False, False
- 4: False, True

► Answer

 [Back to Top](#)

17. What is the output of below code

```
console.log(10 + '10');  
console.log(10 - '10');
```

- 1: 20, 0
- 2: 1010, 0
- 3: 1010, 10-10
- 4: NaN, NaN

► Answer

 [Back to Top](#)

18. What is the output of below code

```
console.log([0] == false);
if([0]) {
  console.log("I'm True");
} else {
  console.log("I'm False");
}
```

- 1: True, I'm True
- 2: True, I'm False
- 3: False, I'm True
- 4: False, I'm False

► [Answer](#)

19. What is the output of below code

```
console.log([1, 2] + [3, 4]);
```

- 1: [1,2,3,4]
- 2: [1,2][3,4]
- 3: SyntaxError
- 4: 1,23,4

► [Answer](#)

 [Back to Top](#)

20. What is the output of below code

```
const numbers = new Set([1, 1, 2, 3, 4]);
console.log(numbers);

const browser = new Set('Firefox');
console.log(browser);
```

- 1: {1, 2, 3, 4}, {"F", "i", "r", "e", "f", "o", "x"}
- 2: {1, 2, 3, 4}, {"F", "i", "r", "e", "o", "x"}
- 3: [1, 2, 3, 4], ["F", "i", "r", "e", "o", "x"]

- 4: {1, 1, 2, 3, 4}, {"F", "i", "r", "e", "f", "o", "x"}

► Answer

 [Back to Top](#)

21. What is the output of below code

```
console.log(NaN === NaN);
```

- 1: True
- 2: False

► Answer

 [Back to Top](#)

22. What is the output of below code

```
let numbers = [1, 2, 3, 4, NaN];
console.log(numbers.indexOf(NaN));
```

- 1: 4
- 2: NaN
- 3: SyntaxError
- 4: -1

► Answer

 [Back to Top](#)

23. What is the output of below code

```
let [a, ...b,] = [1, 2, 3, 4, 5];
console.log(a, b);
```

- 1: 1, [2, 3, 4, 5]
- 2: 1, {2, 3, 4, 5}
- 3: SyntaxError
- 4: 1, [2, 3, 4]

► Answer

 [Back to Top](#)

25. What is the output of below code

```
async function func() {  
    return 10;  
}  
console.log(func());
```

- 1: Promise {: 10}
- 2: 10
- 3: SyntaxError
- 4: Promise {: 10}

► Answer

 [Back to Top](#)

26. What is the output of below code

```
async function func() {  
    await 10;  
}  
console.log(func());
```

- 1: Promise {: 10}
- 2: 10
- 3: SyntaxError
- 4: Promise {: undefined}

► Answer

 [Back to Top](#)

27. What is the output of below code

```
function delay() {  
    return new Promise(resolve => setTimeout(resolve, 2000));  
}
```

```

async function delayedLog(item) {
  await delay();
  console.log(item);
}

async function processArray(array) {
  array.forEach(item => {
    await delayedLog(item);
  })
}

processArray([1, 2, 3, 4]);

```

- 1: SyntaxError
- 2: 1, 2, 3, 4
- 3: 4, 4, 4, 4
- 4: 4, 3, 2, 1

► Answer

[↑ Back to Top](#)

28. What is the output of below code

```

function delay() {
  return new Promise(resolve => setTimeout(resolve, 2000));
}

async function delayedLog(item) {
  await delay();
  console.log(item);
}

async function process(array) {
  array.forEach(async (item) => {
    await delayedLog(item);
  });
  console.log('Process completed!');
}

process([1, 2, 3, 5]);

```

- 1: 1 2 3 5 and Process completed!
- 2: 5 5 5 5 and Process completed!
- 3: Process completed! and 5 5 5 5
- 4: Process completed! and 1 2 3 5

► Answer

 [Back to Top](#)

29. What is the output of below code

```
var set = new Set();
set.add("+0").add("-0").add(NaN).add(undefined).add(NaN);
console.log(set);
```

- 1: Set(4) {"+0", "-0", NaN, undefined}
- 2: Set(3) {"+0", NaN, undefined}
- 3: Set(5) {"+0", "-0", NaN, undefined, NaN}
- 4: Set(4) {"+0", NaN, undefined, NaN}

► Answer

 [Back to Top](#)

30. What is the output of below code

```
const sym1 = Symbol('one');
const sym2 = Symbol('one');

const sym3 = Symbol.for('two');
const sym4 = Symbol.for('two');

console.log(sym1 === sym2, sym3 === sym4);
```

- 1: true, true
- 2: true, false
- 3: false, true
- 4: false, false

► Answer

 [Back to Top](#)

31. What is the output of below code

```
const sym1 = new Symbol('one');
```

```
console.log(sym1);
```

- 1: SyntaxError
- 2: one
- 3: Symbol('one')
- 4: Symbol

► Answer

 [Back to Top](#)

32. What is the output of below code

```
let myNumber = 100;
let myString = '100';

if (!typeof myNumber === "string") {
    console.log("It is not a string!");
} else {
    console.log("It is a string!");
}

if (!typeof myString === "number"){
    console.log("It is not a number!")
} else {
    console.log("It is a number!");
}
```

- 1: SyntaxError
- 2: It is not a string!, It is not a number!
- 3: It is not a string!, It is a number!
- 4: It is a string!, It is a number!

► Answer

 [Back to Top](#)

33. What is the output of below code

```
console.log(JSON.stringify({ myArray: ['one', undefined, function(){}], Symbol('') } ))
console.log(JSON.stringify({ [Symbol.for('one')]: 'one' }, [Symbol.for('one')]));
```

- 1: {"myArray":['one', undefined, {}, Symbol]}, {}

- 2: {"myArray":['one', null,null,null]}, {}
- 3: {"myArray":['one', null,null,null]}, "{ [Symbol.for('one')]: 'one' }, [Symbol.for('one')]"
- 4: {"myArray":['one', undefined, function(){}, Symbol()]}, {}

► Answer

 [Back to Top](#)

34. What is the output of below code

```
class A {  
    constructor() {  
        console.log(new.target.name)  
    }  
}  
  
class B extends A { constructor() { super() } }  
  
new A();  
new B();
```

- 1: A, A
- 2: A, B

► Answer

 [Back to Top](#)

35. What is the output of below code

```
const [x, ...y,] = [1, 2, 3, 4];  
console.log(x, y);
```

- 1: 1, [2, 3, 4]
- 2: 1, [2, 3]
- 3: 1, [2]
- 4: SyntaxError

► Answer

 [Back to Top](#)

36. What is the output of below code

```
const {a: x = 10, b: y = 20} = {a: 30};  
  
console.log(x);  
console.log(y);
```

- 1: 30, 20
- 2: 10, 20
- 3: 10, undefined
- 4: 30, undefined

► Answer

 [Back to Top](#)

37. What is the output of below code

```
function area({length = 10, width = 20}) {  
    console.log(length*width);  
}  
  
area();
```

- 1: 200
- 2: Error
- 3: undefined
- 4: 0

► Answer

 [Back to Top](#)

38. What is the output of below code

```
const props = [
  { id: 1, name: 'John' },
  { id: 2, name: 'Jack' },
  { id: 3, name: 'Tom' }
];

const [,, { name }] = props;
console.log(name);
```

- 1: Tom
- 2: Error
- 3: undefined
- 4: John

► Answer

 [Back to Top](#)

39. What is the output of below code

```
function checkType(num = 1) {
  console.log(typeof num);
}

checkType();
checkType(undefined);
checkType('');
checkType(null);
```

- 1: number, undefined, string, object
- 2: undefined, undefined, string, object
- 3: number, number, string, object
- 4: number, number, number, number

► Answer

 [Back to Top](#)

40. What is the output of below code

```
function add(item, items = []) {
  items.push(item);
  return items;
```

```
}
```

```
console.log(add('Orange'));
console.log(add('Apple'));
```

- 1: ['Orange'], ['Orange', 'Apple']
- 2: ['Orange'], ['Apple']

► Answer

[↑ Back to Top](#)

41. What is the output of below code

```
function greet(greeting, name, message = greeting + ' ' + name) {
  console.log([greeting, name, message]);
}

greet('Hello', 'John');
greet('Hello', 'John', 'Good morning!');
```

- 1: SyntaxError
- 2: ['Hello', 'John', 'Hello John'], ['Hello', 'John', 'Good morning!']

► Answer

[↑ Back to Top](#)

42. What is the output of below code

```
function outer(f = inner()) {
  function inner() { return 'Inner' }
}
outer();
```

- 1: ReferenceError
- 2: Inner

► Answer

[↑ Back to Top](#)

43. What is the output of below code

```
function myFun(x, y, ...manyMoreArgs) {  
    console.log(manyMoreArgs)  
}  
  
myFun(1, 2, 3, 4, 5);  
myFun(1, 2);
```

- 1: [3, 4, 5], undefined
- 2: SyntaxError
- 3: [3, 4, 5], []
- 4: [3, 4, 5], [undefined]

► Answer

 [Back to Top](#)

44. What is the output of below code

```
const obj = {'key': 'value'};  
const array = [...obj];  
console.log(array);
```

- 1: ['key', 'value']
- 2: TypeError
- 3: []
- 4: ['key']

► Answer

 [Back to Top](#)

45. What is the output of below code

```
function* myGenFunc() {  
    yield 1;  
    yield 2;  
    yield 3;  
}  
var myGenObj = new myGenFunc;  
console.log(myGenObj.next().value);
```

- 1: 1
- 2: undefined
- 3: SyntaxError
- 4: TypeError

► Answer

[↑ Back to Top](#)

46. What is the output of below code

```
function* yieldAndReturn() {
  yield 1;
  return 2;
  yield 3;
}

var myGenObj = yieldAndReturn()
console.log(myGenObj.next());
console.log(myGenObj.next());
console.log(myGenObj.next());
```

- 1: { value: 1, done: false }, { value: 2, done: true }, { value: undefined, done: true }
- 2: { value: 1, done: false }, { value: 2, done: false }, { value: undefined, done: true }
- 3: { value: 1, done: false }, { value: 2, done: true }, { value: 3, done: true }
- 4: { value: 1, done: false }, { value: 2, done: false }, { value: 3, done: true }

► Answer

[↑ Back to Top](#)

Releases

47. What is the output of below code

No releases published

```
const myGenerator = (function *(){
  yield 1;
  yield 2;
  yield 3;
```

Packages
No packages published

```
for (const value of myGenerator) {
  console.log(value);
  break;
```

Contributors
26



```
console.log(value);
```

+ 15 contributors

- Languages
- 1: 1,2,3 and 1,2,3
 - 2: 1,2,3 and 4,5,6

- 3: 1 and 1
- 4: 1

► Answer

 [Back to Top](#)

48. What is the output of below code

```
const num = 0o38;  
console.log(num);
```

- 1: SyntaxError
- 2: 38

► Answer

 [Back to Top](#)

49. What is the output of below code

```
const squareObj = new Square(10);  
console.log(squareObj.area);  
  
class Square {  
    constructor(length) {  
        this.length = length;  
    }  
  
    get area() {  
        return this.length * this.length;  
    }  
  
    set area(value) {  
        this.area = value;  
    }  
}
```

- 1: 100

- 2: ReferenceError

► Answer

 [Back to Top](#)

50. What is the output of below code

```
function Person() { }

Person.prototype.walk = function() {
    return this;
}

Person.run = function() {
    return this;
}

let user = new Person();
let walk = user.walk;
console.log(walk());

let run = Person.run;
console.log(run());
```

- 1: undefined, undefined
- 2: Person, Person
- 3: SyntaxError
- 4: Window, Window

► Answer

 [Back to Top](#)

51. What is the output of below code

```
class Vehicle {
    constructor(name) {
        this.name = name;
    }

    start() {
        console.log(` ${this.name} vehicle started`);
    }
}
```

```
class Car extends Vehicle {  
    start() {  
        console.log(` ${this.name} car started`);  
        super.start();  
    }  
}  
  
const car = new Car('BMW');  
console.log(car.start());
```

- 1: SyntaxError
- 2: BMW vehicle started, BMW car started
- 3: BMW car started, BMW vehicle started
- 4: BMW car started, BMW car started

► Answer

 [Back to Top](#)

52. What is the output of below code

```
const USER = {'age': 30};  
USER.age = 25;  
console.log(USER.age);
```

- 1: 30
- 2: 25
- 3: Uncaught TypeError
- 4: SyntaxError

► Answer

 [Back to Top](#)

53. What is the output of below code

```
console.log('😊' === '😊');
```

- 1: false
- 2: true

► Answer

 [Back to Top](#)

54. What is the output of below code?

```
console.log(typeof typeof typeof true);
```

- 1: string
- 2: boolean
- 3: NaN
- 4: number

► [Answer](#)

 [Back to Top](#)

55. What is the output of below code?

```
let zero = new Number(0);

if (zero) {
    console.log("If");
} else {
    console.log("Else");
}
```

- 1: If
- 2: Else
- 3: NaN
- 4: SyntaxError

► [Answer](#)

 [Back to Top](#)

55. What is the output of below code in non strict mode?

```
let msg = "Good morning!!";  
  
msg.name = "John";  
  
console.log(msg.name);
```

- 1: ""
- 2: Error
- 3: John
- 4: Undefined

► Answer

 [Back to Top](#)