**General React – React Interview Questions**

If you know JavaScript, React Native will be easy to pick-up, allowing most front-end web developer to be a mobile developer. All you need to know is JavaScript, platform APIs, some native UI elements, and any other platform-specific design patterns and you’re set.

**General React – React Interview Questions**

**1.  Differentiate between Real DOM and Virtual DOM.**

|  |  |
| --- | --- |
| Real DOM vs Virtual DOM | |
| **Real DOM** | **Virtual  DOM** |
| 1. It updates slow. | 1. It updates faster. |
| 2. Can directly update HTML. | 2. Can’t directly update HTML. |
| 3. Creates a new DOM if element updates. | 3. Updates the JSX if element updates. |
| 4. DOM manipulation is very expensive. | 4. DOM manipulation is very easy. |
| 5. Too much of memory wastage. | 5. No memory wastage. |

**2. What is React?**

* React is a front-end JavaScript library developed by Facebook in 2011.
* It follows the component based approach which helps in building reusable UI components.
* It is used for developing complex and interactive web and mobile UI.
* Even though it was open-sourced only in 2015, it has one of the largest communities supporting it.

**3. What are the features of React?**

Major features of React are listed below:

1. It uses the **virtual DOM** instead of the real DOM.
2. It uses **server-side rendering**.
3. It follows **uni-directional data flow** or data binding.

### ****4. List some of the major advantages of React.****

Some of the major advantages of React are:

1. It increases the application’s performance
2. It can be conveniently used on the client as well as server side
3. Because of JSX, code’s readability increases
4. React is easy to integrate with other frameworks like Meteor, Angular, etc
5. Using React, writing UI test cases become extremely easy

### ****5. What are the limitations of React?****

Limitations of React are listed below:

1. React is just a library, not a full-blown framework
2. Its library is very large and takes time to understand
3. It can be little difficult for the novice programmers to understand
4. Coding gets complex as it uses inline templating and JSX

**4. What are the limitations of React?**

1. Limitations of React are listed below:
2. React is just a library, not a full-blown framework - Correct  
   Its library is very large and takes time to understand - Well library is simple not very large as compare to Angular and Ember  
   It can be little difficult for the novice programmers to understand - its better then Ember and Angular ,Also React dcumentation is comprehensively structured  
   Coding gets complex as it uses inline templating and JSX - Coding standards can be implemented to have complexity divided to modularity

### ****6. What is JSX?****

JSX is a shorthand for JavaScript XML. This is a type of file used by React which utilizes the expressiveness of JavaScript along with HTML like template syntax. This makes the HTML file really easy to understand. This file makes applications robust and boosts its performance. Below is an example of JSX:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | render(){      return(    <div>    <h1> Hello World from Edureka!!</h1>             </div>        );  } |

### ****7. What do you understand by Virtual DOM? Explain its working.****

A virtual DOM is a lightweight JavaScript object which originally is just the copy of the real DOM. It is a node tree that lists the elements, their attributes and content as Objects and their properties. React’s render function creates a node tree out of the React components. It then updates this tree in response to the mutations in the data model which is caused by various actions done by the user or by the system.  
This Virtual DOM works in three simple steps.

1. Whenever any underlying data changes, the entire UI is re-rendered in Virtual DOM representation.
2. Then the difference between the previous DOM representation and the new one is calculated.
3. Once the calculations are done, the real DOM will be updated with only the things that have actually changed. 

### ****8. Why can’t browsers read JSX?****

Browsers can only read JavaScript objects but JSX in not a regular JavaScript object. Thus to enable a browser to read JSX, first, we need to transform JSX file into a JavaScript object using JSX transformers like Babel and then pass it to the browser.

### ****9. How different is React’s ES6 syntax when compared to ES5?****

Syntax has changed from ES5 to ES6 in following aspects:

1. require vs import

|  |  |
| --- | --- |
| 1  2  3  4  5 | // ES5  var React = require('react');    // ES6  import React from 'react'; |

1. export vs exports

|  |  |
| --- | --- |
| 1  2  3  4  5 | // ES5  module.exports = Component;    // ES6  export default Component; |

1. component and function

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | // ES5  var MyComponent = React.createClass({      render: function() {          return    <h3>Hello Edureka!</h3>  ;      }  });    // ES6  class MyComponent extends React.Component {      render() {          return    <h3>Hello Edureka!</h3>  ;      }  } |

1. props

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | // ES5  var App = React.createClass({      propTypes: { name: React.PropTypes.string },      render: function() {          return    <h3>Hello, {this.props.name}!</h3>  ;      }  });    // ES6  class App extends React.Component {      render() {          return    <h3>Hello, {this.props.name}!</h3>  ;      }  } |

1. state

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | // ES5  var App = React.createClass({      getInitialState: function() {          return { name: 'world' };      },      render: function() {          return    <h3>Hello, {this.state.name}!</h3>  ;      }  });    // ES6  class App extends React.Component {      constructor() {          super();          this.state = { name: 'world' };      }      render() {          return    <h3>Hello, {this.state.name}!</h3>  ;      }  } |

### ****10. How is React different from Angular?****

|  |  |  |
| --- | --- | --- |
| React vs Angular | | |
| **TOPIC** | **REACT** | **ANGULAR** |
| 1. ARCHITECTURE | Only the View of MVC | Complete MVC |
| 2. RENDERING | Server-side rendering | Client-side rendering |
| 3. DOM | Uses virtual DOM | Uses real DOM |
| 4. DATA BINDING | One-way data binding | Two-way data binding |
| 5. DEBUGGING | Compile time debugging | Runtime debugging |
| 6. AUTHOR | Facebook | Google |

**React Components – React Interview Questions**

### ****11. What do you understand from “In React, everything is a component.”****

Components are the building blocks of a React application’s UI. These components split up the entire UI into small independent and reusable pieces. Then it renders each of these components independent of each other without affecting the rest of the UI.

### ****12. 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.

### ****13. How can you embed two or more components into one?****

We can embed components into one in the following way:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | class MyComponent extends React.Component{      render(){          return(    <div>    <h1>Hello</h1>                    <Header/>              </div>            );      }  }  class Header extends React.Component{      render(){          return    <h1>Header Component</h1>       };  }  ReactDOM.render(      <MyComponent/>, document.getElementById('content')  ); |

### ****14. What is Props?****

Props is the shorthand for Properties in React. They are read-only components which must be kept pure i.e. immutable. They are always passed down from the parent to the child components throughout the application. A child component can never send a prop back to the parent component. This help in maintaining the unidirectional data flow and are generally used to render the dynamically generated data.

### ****15. What is a state in React and how is it used?****

States are the heart of React components. States are the source of data and must be kept as simple as possible. Basically, states are the objects which determine components rendering and behavior. They are mutable unlike the props and create dynamic and interactive components. They are accessed via **this.state().**

### ****16. Differentiate between states and props.****

|  |  |  |
| --- | --- | --- |
| States vs Props | | |
| **Conditions** | **State** | **Props** |
| 1. Receive initial value from parent component | Yes | Yes |
| 2. Parent component can change value | No | Yes |
| 3. Set default values inside component | Yes | Yes |
| 4. Changes inside component | Yes | No |
| 5. Set initial value for child components | Yes | Yes |
| 6. Changes inside child components | No | Yes |

### ****17. How can you update the state of a component?****

State of a component can be updated using this.setState().

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | class MyComponent extends React.Component {      constructor() {          super();          this.state = {              name: 'Maxx',              id: '101'          }      }      render()          {              setTimeout(()=>{this.setState({name:'Jaeha', id:'222'})},2000)              return (    <div>    <h1>Hello {this.state.name}</h1>    <h2>Your Id is {this.state.id}</h2>                       </div>                );          }      }  ReactDOM.render(      <MyComponent/>, document.getElementById('content')  ); |

### ****18. 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.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | //General way  render() {      return(          <MyInput onChange={this.handleChange.bind(this) } />      );  }  //With Arrow Function  render() {      return(          <MyInput onChange={ (e) => this.handleOnChange(e) } />      );  } |

### ****19. Differentiate between stateful and stateless components.****

|  |  |
| --- | --- |
| Stateful vs Stateless | |
| **Stateful Component** | **Stateless Component** |
| 1. Stores info about component’s state change in memory | 1. Calculates the internal state of the components |
| 2. Have authority to change state | 2. Do not have the authority to change state |
| 3. Contains the knowledge of past, current and possible future changes in state | 3. Contains no knowledge of past, current and possible future state changes |
| 4. Stateless components notify them about the requirement of the state change, then they send down the props to them. | 4. They receive the props from the Stateful components and treat them as callback functions. |

### ****20. What are the different phases of React component’s lifecycle?****

There are three different phases of React component’s lifecycle:

1. Initial Rendering Phase: This is the phase when the component is about to start its life journey and make its way to the DOM.
2. 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.
3. Unmounting Phase: This is the final phase of a component’s life cycle in which the component is destroyed and removed from the DOM.

### ****21. Explain the lifecycle methods of React components in detail.****

Some of the most important lifecycle methods are:

1. **componentWillMount()**–Executed just before rendering takes place both on the client as well as server-side.
2. **componentDidMount()**–Executed on the client side only after the first render.
3. **componentWillReceiveProps()**– Invoked as soon as the props are received from the parent class and before another render is called.
4. **shouldComponentUpdate()**–Returns true or false value based on certain conditions. If you want your component to update, return **true** else return **false**. By default, it returns false.
5. **componentWillUpdate()**– Called just before rendering takes place in the DOM.
6. **componentDidUpdate()**–Called immediately after rendering takes place.
7. **componentWillUnmount()**– Called after the component is unmounted from the DOM. It is used to clear up the memory spaces.

### ****22. What is an event in React?****

In React, events are the triggered reactions to specific actions like mouse hover, mouse click, key press, etc. Handling these events are similar to handling events in DOM elements. But there are some syntactical differences like:

1. Events are named using camel case instead of just using the lowercase.
2. Events are passed as functions instead of strings.

The event argument contains a set of properties, which are specific to an event. Each event type contains its own properties and behavior which can be accessed via its event handler only.

### ****23. How do you create an event in React?****

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | class Display extends React.Component({      show(evt) {          // code      },      render() {          // Render the div with an onClick prop (value is a function)          return (    <div onClick={this.show}>Click Me!</div>            );      }  }); |

### ****24. 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.

### ****25. What do you understand by 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.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | class ReferenceDemo extends React.Component{       display() {           const name = this.inputDemo.value;           document.getElementById('disp').innerHTML = name;       }  render() {      return(    <div>              Name: <input type="text" ref={input => this.inputDemo = input} />              <button name="Click" onClick={this.display}>Click</button>    <h2>Hello <span id="disp"></span> !!!</h2>          </div>      );     }   } |

### ****26. List some of the cases when you should use Refs.****

Following are the cases when refs should be used:

* When you need to manage focus, select text or media playback
* To trigger imperative animations
* Integrate with third-party DOM libraries

### ****27. How do you modularize code in React?****

We can modularize code by using the export and import properties. They help in writing the components separately in different files.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | //ChildComponent.jsx  export default class ChildComponent extends React.Component {      render() {          return(    <div>    <h1>This is a child component</h1>               </div>            );      }  }    //ParentComponent.jsx  import ChildComponent from './childcomponent.js';  class ParentComponent extends React.Component {      render() {          return(    <div>                  <App />              </div>            );      }  } |

### ****28. How**** ****are forms created in React?****

React forms are similar to HTML forms. But in React, the state is contained in the state property of the component and is only updated via setState(). Thus the elements can’t directly update their state and their submission is handled by a JavaScript function. This function has full access to the data that is entered by the user into a form.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | handleSubmit(event) {      alert('A name was submitted: ' + this.state.value);      event.preventDefault();  }    render() {      return (    <form onSubmit={this.handleSubmit}>              <label>                  Name:                  <input type="text" value={this.state.value} onChange={this.handleSubmit} />              </label>              <input type="submit" value="Submit" />          </form>        );  } |

### ****29. What do you know about controlled and uncontrolled components?****

|  |  |
| --- | --- |
| Controlled vs Uncontrolled Components | |
| **Controlled Components** | **Uncontrolled Components** |
| 1. They do not maintain their own state | 1. They maintain their own state |
| 2. Data is controlled by the parent component | 2. Data is controlled by the DOM |
| 3. They take in the current values through props and then notify the changes via callbacks | 3. Refs are used to get their current values |

### ****30. 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.

### ****31. What can you do with HOC?****

HOC can be used for many tasks like:

* Code reuse, logic and bootstrap abstraction
* Render High jacking
* State abstraction and manipulation
* Props manipulation

### ****32. What are Pure Components?****

*Pure*components are the simplest and fastest components which can be written. They can replace any component which only has a **render().**These components enhance the simplicity of the code and performance of the application.

### ****33. What is the significance of keys in React?****

Keys are used for identifying unique Virtual DOM Elements with their corresponding data driving the UI. They help React to optimize the rendering by recycling all the existing elements in the DOM. These keys must be a unique number or string, using which React just reorders the elements instead of re-rendering them. This leads to increase in application’s performance.

**React Redux – React Interview Questions**

### ****34. What were the major problems with MVC framework?****

Following are some of the major problems with MVC framework:

* DOM manipulation was very expensive
* Applications were slow and inefficient
* There was huge memory wastage
* Because of circular dependencies, a complicated model was created around models and views

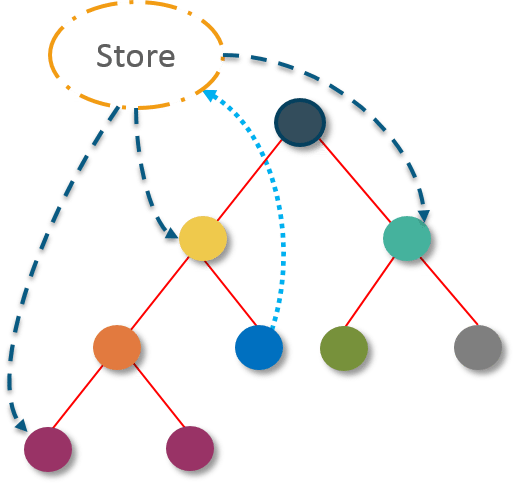
### ****35. Explain Flux.****

Flux is an architectural pattern which enforces the uni-directional data flow. It controls derived data and enables communication between multiple components using a central Store which has authority for all data. Any update in data throughout the application must occur here only. Flux provides stability to the application and reduces run-time errors.

### ****36.**** ****What is Redux?****

Redux is one of the hottest libraries for front-end development in today’s marketplace. It is a predictable state container for JavaScript applications and is used for the entire applications state management. Applications developed with Redux are easy to test and can run in different environments showing consistent behavior.

### ****37. What are the three principles that Redux follows?****

1. ***Single source of truth:***The state of the entire application is stored in an object/ state 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.
2. **State is read-only:**The only way to change the state is to trigger an action. An action is a plain JS object describing the change. Just like state is the minimal representation of data, the action is the minimal representation of the change to that data.
3. ***Changes are made with pure functions:*** In order to specify how the state tree is transformed by actions, you need pure functions. Pure functions are those whose return value depends solely on the values of their arguments.

### ****38. What do you understand by “Single source of truth”?****

Redux uses ‘Store’ for storing the application’s entire state at one place. So all the component’s state are stored in the Store and they receive updates from the Store itself. The single state tree makes it easier to keep track of changes over time and debug or inspect the application.

### ****39. List down the components of Redux.****

Redux is composed of the following components:

1. **Action** – It’s an object that describes what happened.
2. **Reducer**–  It is a place to determine how the state will change.
3. **Store** – State/ Object tree of the entire application is saved in the Store.
4. **View** – Simply displays the data provided by the Store.

### ****40. Show how the data flows through Redux?****



### ****41. How are Actions defined in Redux?****

Actions in React must have a type property that indicates the type of ACTION being performed. They must be defined as a String constant and you can add more properties to it as well. In Redux, actions are created using the functions called Action Creators. Below is an example of Action and Action Creator:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | function addTodo(text) {         return {                  type: ADD\_TODO,                   text      }  } |

### ****42. Explain the role of Reducer.****

Reducers are pure functions which specify how the application’s state changes in response to an ACTION. Reducers work by taking in the previous state and action, and then it returns a new state. It determines what sort of update needs to be done based on the type of the action, and then returns new values. It returns the previous state as it is, if no work needs to be done.

### ****43. What is the significance of Store in Redux?****

A store is a JavaScript object which can hold the application’s state and provide a few helper methods to access the state, dispatch actions and register listeners. The entire state/ object tree of an application is saved in a single store. As a result of this, Redux is very simple and predictable. We can pass middleware to the store to handle the processing of data as well as to keep a log of various actions that change the state of stores. All the actions return a new state via reducers.

### ****44. How is Redux different from Flux?****

|  |  |
| --- | --- |
| Flux vs Redux | |
| **Flux** | **Redux** |
| 1. The Store contains state and change logic | 1. Store and change logic are separate |
| 2. There are multiple stores | 2. There is only one store |
| 3. All the stores are disconnected and flat | 3. Single store with hierarchical reducers |
| 4. Has singleton dispatcher | 4. No concept of dispatcher |
| 5. React components subscribe to the store | 5. Container components utilize connect |
| 6. State is mutable | 6. State is immutable |

### ****45. What are the advantages of Redux?****

Advantages of Redux are listed below:

* **Predictability of outcome –**Since there is always one source of truth, i.e. the store, there is no confusion about how to sync the current state with actions and other parts of the application.
* **Maintainability –**The code becomes easier to maintain with a predictable outcome and strict structure.
* **Server-side rendering –** You just need to pass the store created on the server, to the client side. This is very useful for initial render and provides a better user experience as it optimizes the application performance.
* **Developer tools –**From actions to state changes, developers can track everything going on in the application in real time.
* **Community and ecosystem –**Redux has a huge community behind it which makes it even more captivating to use. A large community of talented individuals contribute to the betterment of the library and develop various applications with it.
* **Ease of testing –**Redux’s code is mostly functions which are small, pure and isolated. This makes the code testable and independent.
* **Organization –**Redux is precise about how code should be organized, this makes the code more consistent and easier when a team works with it.

**React Router – React Interview Questions**

### ****46. What is React Router?****

React Router is a powerful routing library built on top of React, which helps in adding new screens and flows to the application. This keeps the URL in sync with data that’s being displayed on the web page. It maintains a standardized structure and behavior and is used for developing single page web applications. React Router has a simple API.

### ****47. Why**** ****is switch keyword used in React Router v4?****

Although a **<div>** is used to encapsulate multiple routes inside the Router. The ‘switch’ keyword is used when you want to display only a single route to be rendered amongst the several defined routes. The **<switch>**tag when in use matches the typed URL with the defined routes in sequential order. When the first match is found, it renders the specified route. Thereby bypassing the remaining routes.

### ****48. Why do we need a Router in React?****

A Router is used to define multiple routes and when a user types a specific URL, if this URL matches the path of any ‘route’ defined inside the router, then the user is redirected to that particular route. So basically, we need to add a Router library to our app that allows creating multiple routes with each leading to us a unique view.

|  |  |
| --- | --- |
| 1  2  3  4  5 | <switch>      <route exact path=’/’ component={Home}/>      <route path=’/posts/:id’ component={Newpost}/>      <route path=’/posts’   component={Post}/>  </switch> |

### ****49. List down the advantages of React Router.****

Few advantages are:

1. 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 (**<BrowserRouter>**) in which we enclose the specific child routes (**<route>**).
2. No need to manually set History value: In React Router v4, all we need to do is wrap our routes within the **<BrowserRouter>** component.
3. 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.

### ****50. How is React Router different from conventional routing?****

|  |  |  |
| --- | --- | --- |
| Conventional Routing vs React Routing | | |
| **Topic** | **Conventional Routing** | **React Routing** |
| **PAGES INVOLVED** | Each view corresponds to a new file | Only single HTML page is involved |
| **URL CHANGES** | A HTTP request is sent to a server and corresponding HTML page is received | Only the History attribute is changed |
| **FEEL** | User actually navigates across different pages for each view | User is duped thinking he is navigating across different pages |

More questions :

1) Render Prop Pattern  
2) Fiber Reconciller  
3) Whats better Redux vs Mobx vs Flow  
4) Redux Sagas / Observables and What are Middlewares  
5) Why is Immutability so Important

On Mounting methods and there order are as below:----  
constructor()  
static getDerivedStateFromProps()  
render()  
componentDidMount()

on Update methods and there order are as below :----  
static getDerivedStateFromProps()  
shouldComponentUpdate()  
render()  
getSnapshotBeforeUpdate()  
componentDidUpdate()

on unMounting :---  
componentWillUnmount()

error handling :---

static getDerivedStateFromError()  
componentDidCatch()

and those life cycle hooks which are removed can be used using UNSAFE\_ prefix and its legacy to use.

# ReactDOM

If you load React from a <script> tag, these top-level APIs are available on the ReactDOM global. If you use ES6 with npm, you can write import ReactDOM from 'react-dom'. If you use ES5 with npm, you can write var ReactDOM = require('react-dom').

The react-dom package provides DOM-specific methods that can be used at the top level of your app and as an escape hatch to get outside of the React model if you need to. Most of your components should not need to use this module.

* [render()](https://reactjs.org/docs/react-dom.html#render)
* [hydrate()](https://reactjs.org/docs/react-dom.html#hydrate)
* [unmountComponentAtNode()](https://reactjs.org/docs/react-dom.html#unmountcomponentatnode)
* [findDOMNode()](https://reactjs.org/docs/react-dom.html#finddomnode)
* [createPortal()](https://reactjs.org/docs/react-dom.html#createportal)

**ReactDOM.render(element, container[, callback])**

ReactDOM.render() controls the contents of the container node you pass in. Any existing DOM elements inside are replaced when first called. Later calls use React’s DOM diffing algorithm for efficient updates.

ReactDOM.render() does not modify the container node (only modifies the children of the container). It may be possible to insert a component to an existing DOM node without overwriting the existing children.

ReactDOM.render() currently returns a reference to the root ReactComponent instance. However, using this return value is legacy and should be avoided because future versions of React may render components asynchronously in some cases. If you need a reference to the root ReactComponent instance, the preferred solution is to attach a [callback ref](https://reactjs.org/docs/more-about-refs.html#the-ref-callback-attribute) to the root element.

Using ReactDOM.render() to hydrate a server-rendered container is deprecated and will be removed in React 17. Use [hydrate()](https://reactjs.org/docs/react-dom.html#hydrate) instead.

**ReactDOM.hydrate(element, container[, callback])**

Same as [render()](https://reactjs.org/docs/react-dom.html#render), but is used to hydrate a container whose HTML contents were rendered by [ReactDOMServer](https://reactjs.org/docs/react-dom-server.html). React will attempt to attach event listeners to the existing markup.

React expects that the rendered content is identical between the server and the client. It can patch up differences in text content, but you should treat mismatches as bugs and fix them. In development mode, React warns about mismatches during hydration. There are no guarantees that attribute differences will be patched up in case of mismatches. This is important for performance reasons because in most apps, mismatches are rare, and so validating all markup would be prohibitively expensive.

**ReactDOM.unmountComponentAtNode(container)**

Remove 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.

**ReactDOM.findDOMNode(component)**

**Note:** findDOMNode is an escape hatch used to access the underlying DOM node. In most cases, use of this escape hatch is discouraged because it pierces the component abstraction. [It has been deprecated in StrictMode.](https://reactjs.org/docs/strict-mode.html#warning-about-deprecated-finddomnode-usage)

If this component has been mounted into the DOM, this returns the corresponding native browser DOM element. This method is useful for reading values out of the DOM, such as form field values and performing DOM measurements. **In most cases, you can attach a ref to the DOM node and avoid using findDOMNode at all.**

When a component renders to null or false, findDOMNode returns null. When a component renders to a string, findDOMNode returns a text DOM node containing that value. As of React 16, a component may return a fragment with multiple children, in which case findDOMNode will return the DOM node corresponding to the first non-empty child.

**Note:**

findDOMNode only works on mounted components (that is, components that have been placed in the DOM). If you try to call this on a component that has not been mounted yet (like calling findDOMNode() in render() on a component that has yet to be created) an exception will be thrown.

findDOMNode cannot be used on function components.

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

Creates a portal. Portals provide a way to [render children into a DOM node that exists outside the hierarchy of the DOM component](https://reactjs.org/docs/portals.html).

# 5 Essential React.js Interview Questions and Answers

In the [2017 developer survey](https://insights.stackoverflow.com/survey/2017), Stack Overflow noted that React is still among the the most popular JavaScript libraries to date. React has exploded in popularity because its simple and declarative API produces highly-performant applications — and that momentum only continues to grow.

If you’re looking to build a robust web application, chances are that React may be a good fit for you. Once you're ready to hire a React developer, here are essential interview questions to ask and some advanced concepts to know.

### ****A Word on Technical Interviews****

Before we dive right into the questions, it needs to be said that technical interviews are notorious for gotcha-style questions and irrelevant whiteboarding exercises. This article avoids that interview style entirely — rather, I outlined five general (yet vital) concepts which I believe any seasoned React developer should know.

Over the years, I have been in countless interviews as both the applicant and the conductor. My experience has taught me that the best candidates for an engineering role are those who can **articulate intelligent opinions** and **defend them using examples from their own experience**. Pair-programming relevant examples as a follow-up to discussion would be my preferred interview format, but we will stick to the Q&A portion for this article.

## Question #1: What is React? How is it different from other JS frameworks?

Although this sounds like a relatively simple question, it’s really asking the candidate to state an informed opinion about React, as well as any competing alternatives. In short, this question is designed to test a candidate's knowledge about the JavaScript ecosystem at large while also pressing for specifics on what makes React unique.

Let’s look at each part of the answer separately.

### ****What is React?****

React is an open-source JavaScript library created by Facebook for building complex, interactive UIs in web and mobile applications.

The key point in this answer is that React’s core purpose is to build UI components; it is often referred to as just the “V” (View) in an “MVC” architecture. Therefore it has no opinions on the other pieces of your technology stack and can be [seamlessly integrated into any application](https://www.codementor.io/javascript/tutorial/should-you-build-your-web-application-with-javascript-mvc-frameworks).

### ****How is React different?****

The answer to this question will likely vary depending on the candidate's personal experiences. The important thing is to listen for real-life examples provided and opinions on whether or not the candidate prefers React and why.

Because React is a small library focused on building UI components, it is necessarily different than a lot of other JavaScript frameworks.

For example, AngularJS (1.x) approaches building an application by extending HTML markup and injecting various constructs (e.g. Directives, Controllers, Services) at runtime. As a result, AngularJS is very opinionated about the greater architecture of your application — these abstractions are certainly useful in some cases, but in many situations, they come at the cost of flexibility.

By contrast, React focuses exclusively on the creation of components, and has few (if any) opinions about an application’s architecture. This allows a developer an incredible amount of flexibility in choosing the architecture they deem “best” — though it also places the responsibility of choosing (or building) those parts on the developer.

I recently migrated an application originally written in AngularJS to React, and one of the things I loved most was…

By comparing and contrasting React with another library, not only can the candidate demonstrate a deep understanding of React, but also position themself as a potentially strong candidate.

Be prepared to ask some follow-up questions as well, such as:

* Under what circumstances would you choose React over another technology? For example, [*React vs Angular*](https://www.codementor.io/codementorteam/react-vs-angular-2-comparison-beginners-guide-lvz5710ha) or [*React vs Vue*](https://www.codementor.io/vuejsdevelopers/react-or-vue-which-javascript-ui-library-should-you-be-using-6hri3num4).
* If React only focuses on a small part of building UI components, can you explain some pitfalls one might encounter when developing a large application?
* If you were rewriting an AngularJS application in React, how much code could you expect to re-use?

## Question #2: What happens during the lifecycle of a React component?

One of the most valuable parts of React is its [component lifecycle](https://facebook.github.io/react/docs/component-specs.html) — so understanding exactly how components function over time is instrumental in building a maintainable application.

### ****High-Level Component Lifecycle****

At the highest level, React components have lifecycle events that fall into three general categories:

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

Every React component defines these events as a mechanism for managing its properties, state, and rendered output. Some of these events only happen once, others happen more frequently; understanding these three general categories should help you clearly visualize when certain logic needs to be applied.

For example, a component may need to add event listeners to the DOM when it first mounts. However, it should probably remove those event listeners when the component unmounts from the DOM so that irrelevant processing does not occur.

**class** **MyComponent** **extends** **React**.**Component** {

*// when the component is added to the DOM...*

componentDidMount() {

window.addEventListener('resize', **this**.onResizeHandler);

}

*// when the component is removed from the DOM...*

componentWillUnmount() {

window.removeEventListener('resize', **this**.onResizeHandler);

}

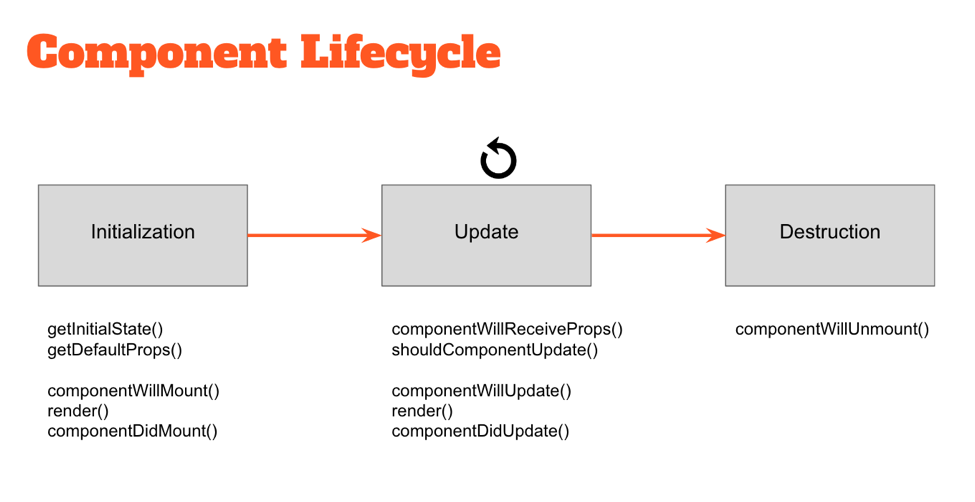
onResizeHandler() {

console.log('The window has been resized!');

}

}

### ****Low-Level Component Lifecycle****



Within these three general buckets exist a number of specific lifecycle hooks — essentially abstract methods — that can be utilized by any React component to more accurately manage updates. Understanding how and when these hooks fire is key to building stable components and will enable you to control the rendering process (improving performance).

Take a look at the diagram above. The events under “Initialization” only happen when a component is first initialized or added to the DOM. Similarly, the events under “Destruction” only happen once (when the component is removed from the DOM). However, the events under “Update” happen every time the properties or state of the component change.

For example, components will automatically re-render themselves any time their properties or state change. However, in some cases a component might not need to update — so preventing the component from re-rendering might improve the performance of our application.

**class** **MyComponent** **extends** **React**.**Component** {

*// only re-render if the ID has changed!*

shouldComponentUpdate(nextProps, nextState) {

**return** nextProps.id === **this**.props.id;

}

}

[](https://www.codementor.io/reactjs-developers)

#### Find top React developers today.

CodementorX will find you the best engineers for your project.

## Question #3: What can you tell me about JSX?

When Facebook first released React to the world, they also introduced a new dialect of JavaScript called JSX that embeds raw HTML templates inside JavaScript code. JSX code by itself cannot be read by the browser; it must be transpiled into traditional JavaScript using tools like Babel and webpack. While many developers understandably have initial knee-jerk reactions against it, JSX (in tandem with ES2015) has become the defacto method of defining React components.

**class** **MyComponent** **extends** **React**.**Component** {

render() {

**let** props = **this**.props;

**return** (

<div className="my-component">

<a href={props.url}>{props.name}</a>

</div>

);

}

}

Asking questions about JSX tests whether or not the candidate can state an informed opinion towards JSX and defend it based on personal experience. Let’s cover some of the basic talking points.

### ****Key Talking Points****

Developers do not have to use JSX (and ES2015) to write an application in React.

This is certainly true. Having said that, many React developers prefer to use JSX as its syntax is far more declarative and reduces overall code complexity. Facebook certainly encourages it in all of their documentation!

Adopting JSX allows the developer to simultaneously adopt ES2015 — giving immediate access to some wonderful syntactic sugar.

ES2015 introduced a variety of new features to JavaScript that makes writing large applications far easier than ever before: [classes](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes), block scoping via [let](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let), and the new [spread](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator) operator are just a small portion of the additions.

**import** AnotherClass **from** './AnotherClass';

**class** **MyComponent** **extends** **React**.**Component** {

render() {

**let** props = **this**.props;

**return** (

<div className="my-component">

<AnotherClass {...props} />

</div>

);

}

}

But while ES2015 is becoming more and more widespread, it still is far from widely supported by the major browsers — so tools like Babel or webpack are needed to convert everything into legacy ES5 code.

Candidates that have built a React application using JSX and ES2015 can speak about some specific pros or cons encountered, such as:

Although it took me some time to get used to the JSX and ES2015 syntax, I discovered how much I really enjoyed using it. Specifically, I’m a big fan of…

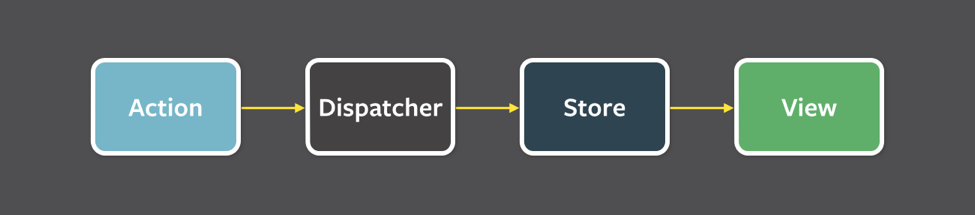
On the other hand, I could do without the hassle of configuring webpack and Babel. Our team ran into issues with…

The React docs on [JSX Gotchas](https://facebook.github.io/react/docs/jsx-gotchas.html) may be good to know/review.

## Question #4: Are you familiar with Flux?

Flux is an architectural pattern that enforces unidirectional data flow — its core purpose is to control derived data so that multiple components can interact with that data without risking pollution.

The Flux pattern is generic; it’s not specific to React applications, nor is it required to build a React app. However, Flux is commonly used by React developers because React components are declarative — the rendered UI (View) is simply a function of state (Store data).



Flux is relatively simple in concept, but in a technical interview, it's important that the developer demonstrates a deep understanding of its implementation. Let’s cover of the important few discussion points.

### ****Description of Flux****

In the Flux pattern, the Store is the central authority for all data; any mutations to the data must occur within the store. Changes to the Store data are subsequently broadcast to subscribing Views via events. Views then update themselves based on the new state of received data.

To request changes to any Store data, Actions may be fired. These Actions are controlled by a central Dispatcher; Actions may not occur simultaneously, ensuring that a Store only mutates data once per Action.

The strict unidirectional flow of this Flux pattern enforces data stability, reducing data-related runtime errors throughout an application.

### ****Flux vs MVC****

Traditional MVC patterns have worked well for separating the concerns of data (Model), UI (View) and logic (Controller) — but many web developers have discovered limitations with that approach as applications grow in size. Specifically, MVC architectures frequently encounter two main problems:

* **Poorly defined data flow:** The cascading updates which occur across views often lead to a tangled web of events which is difficult to debug.
* **Lack of data integrity:** Model data can be mutated from anywhere, yielding unpredictable results across the UI.

With the Flux pattern complex UIs no longer suffer from cascading updates; any given React component will be able to reconstruct its state based on the data provided by the store. The flux pattern also enforces data integrity by restricting direct access to the shared data.

During a technical interview, one should discuss the differences between the Flux and MVC design patterns within the context of a specific example:

For example, imagine we have a “master/detail” UI in which the user can select a record from a list (master view) and edit it using an auto-populated form (detail view).

With an MVC architecture, the data contained within the Model is shared between both the master and detail Views. Each of these views might have its own Controller delegating updates between the Model and the View. At any point the data contained within the Model might be updated — and it’s difficult to know where exactly that change occurred. Did it happen in one of the Views sharing that Model, or in one of the Controllers? Because the Model’s data can be mutated by any actor in the application, the risk of data pollution in complex UIs is greater than we’d like.

With a Flux architecture, the Store data is similarly shared between multiple Views. However this data can’t be directly mutated — all of the requests to update the data must pass through the Action > Dispatcher chain first, eliminating the risk of random data pollution. When updates are made to the data, it’s now much easier to locate the code requesting those changes.

### ****Difference with AngularJS (1.x)****

UI components in AngularJS typically rely on some internal $scope to store their data. This data can be directly mutated from within the UI component or anything given access to $scope — a risky situation for any part of the component or greater application which relies on that data.

By contrast, the Flux pattern encourages the use of immutable data. Because the store is the central authority on all data, any mutations to that data must occur within the store. The risk of data pollution is greatly reduced.

### ****Testing****

One of the most valuable aspects of applications built on Flux is that their components become incredibly easy to test. Developers can recreate and test the state of any React component by simply updating the store — direct interactions with the UI (with tools like [Selenium](http://www.seleniumhq.org/projects/webdriver/)) are no longer necessary in many cases.

### ****Popular Flux Libraries****

While Flux is a general pattern for enforcing data flow through an application, there exist many implementations from which to choose from. There are nuances between each implementation, as well as specific pros and cons to consider. The candidate should provide examples of real-world experience with using Flux.

For example, the candidate might discuss:

* [Redux](http://redux.js.org/): perhaps the most popular Flux library today.
* [Alt.js](http://alt.js.org/): another popular library for managing data in React applications.

## Question #5: What are stateless components?

If React components are essentially state machines that generate UI markup, then what are stateless components?

Stateless components (a flavor of “reusable” components) are nothing more than pure functions that render DOM based solely on the properties provided to them.

**const** StatelessCmp = props => {

**return** (

<div className="my-stateless-component">

{props.name}: {props.birthday}

</div>

);

};

*// ---*

ReactDOM.render(

<StatelessCmp name="Art" birthday="10/01/1980" />,

document.getElementById('main')

);

This component has no need for any internal state — let alone a constructor or lifecycle handlers. The output of the component is purely a function of the properties provided to it.

## Bonus Question: Explain this Code

As I mentioned at the beginning of this article, [technical interviews](http://skillcrush.com/2016/03/29/rock-your-next-whiteboard-test/) may also include time where the developer is asked to look at (and probably write) some code. Take a look at the code below:

**class** **MyComponent** **extends** **React**.**Component** {

**constructor**(props) {

*// set the default internal state*

**this**.state = {

clicks: 0

};

}

componentDidMount() {

**this**.refs.myComponentDiv.addEventListener('click', **this**.clickHandler);

}

componentWillUnmount() {

**this**.refs.myComponentDiv.removeEventListener('click', **this**.clickHandler);

}

clickHandler() {

**this**.setState({

clicks: **this**.clicks + 1

});

}

render() {

**let** children = **this**.props.children;

**return** (

<div className="my-component" ref="myComponentDiv">

<h2>My Component ({this.state.clicks} clicks})</h2>

<h3>{this.props.headerText}</h3>

{children}

</div>

);

}

}

**Given the code defined above, can you identify two problems?**

1. The constructor does not pass its props to the super class. It should include the following line:

**constructor**(props) {

**super**(props);

*// ...*

}

1. The event listener (when assigned via addEventListener()) is not properly scoped because [ES2015 doesn’t provide autobinding](https://facebook.github.io/react/docs/reusable-components.html#no-autobinding). Therefore the developer can re-assign clickHandler in the constructor to include the correct binding to this:

**constructor**(props) {

**super**(props);

**this**.clickHandler = **this**.clickHandler.bind(**this**);

*// ...*

}

**Can you explain what the output of this class actually does? How would you use it in an application?**

This class creates a <div /> element and attaches a click listener to it. The content of this component includes a <h2 /> element that updates every time the user clicks on the parent <div />, as well as an <h3 /> element containing a provided title and whatever child elements were passed to it.

To use this class, the candidate should import it into another class and use it like this:

<MyComponent headerText="A list of paragraph tags">

<p>First child.</p>

<p>Any other <span>number</span> of children...</p>

</MyComponent>

### Conclusion

Interviewing a React developer involves much more than just testing for React knowledge — you should also ask [questions about JavaScript](https://www.codementor.io/nihantanu/21-essential-javascript-tech-interview-practice-questions-answers-du107p62z) and about other nuances more closely related to the project or job in question.

# 21 Essential JavaScript Interview Questions

## Question 1

### 1. What is the difference between undefined and not defined in JavaScript?

In JavaScript, if you try to use a variable that doesn't exist and has not been declared, then JavaScript will throw an error var name is not defined and script will stop executing. However, if you use typeof undeclared\_variable, then it will return undefined.

Before getting further into this, let's first understand the difference between declaration and definition.

Let's say var x is a declaration because you have not defined what value it holds yet, but you have declared its existence and the need for memory allocation.

> **var** x; *// declaring x*

> console.log(x); *//output: undefined*

Here var x = 1 is both a declaration and definition (also we can say we are doing an initialisation). In the example above, the declaration and assignment of value happen inline for variable x. In JavaScript, every variable or function declaration you bring to the top of its current scope is called hoisting.

The assignment happens in order, so when we try to access a variable that is declared but not defined yet, we will get the result undefined.

**var** x; *// Declaration*

**if**(**typeof** x === 'undefined') *// Will return true*

If a variable that is neither declared nor defined, when we try to reference such a variable we'd get the result not defined.

> console.log(y); *// Output: ReferenceError: y is not defined*

## Question 2

### What will be the output of the code below?

**var** y = 1;

**if** (**function** **f**(){}) {

y += **typeof** f;

}

console.log(y);

The output would be 1undefined. The if condition statement evaluates using eval, so eval(function f(){}) returns function f(){} (which is true). Therefore, inside the if statement, executing typeof f returns undefined because the if statement code executes at run time, and the statement inside the if condition is evaluated during run time.

**var** k = 1;

**if** (1) {

eval(**function** **foo**(){});

k += **typeof** foo;

}

console.log(k);

The code above will also output 1undefined.

**var** k = 1;

**if** (1) {

**function** **foo**(){};

k += **typeof** foo;

}

console.log(k); *// output 1function*

## Question 3

### What is the drawback of creating true private methods in JavaScript?

One of the drawbacks of creating true private methods in JavaScript is that they are very memory-inefficient, as a new copy of the method would be created for each instance.

**var** Employee = **function** (name, company, salary) {

**this**.name = name || ""; *//Public attribute default value is null*

**this**.company = company || ""; *//Public attribute default value is null*

**this**.salary = salary || 5000; *//Public attribute default value is null*

*// Private method*

**var** increaseSalary = **function** () {

**this**.salary = **this**.salary + 1000;

};

*// Public method*

**this**.dispalyIncreasedSalary = **function**() {

increaseSlary();

console.log(**this**.salary);

};

};

*// Create Employee class object*

**var** emp1 = **new** Employee("John","Pluto",3000);

*// Create Employee class object*

**var** emp2 = **new** Employee("Merry","Pluto",2000);

*// Create Employee class object*

**var** emp3 = **new** Employee("Ren","Pluto",2500);

Here each instance variable emp1, emp2, emp3 has its own copy of the increaseSalary private method.

So, as a recommendation, don’t use private methods unless it’s necessary.

## Question 4

### What is a “closure” in JavaScript? Provide an example

A closure is a function defined inside another function (called the parent function), and has access to variables that are declared and defined in the parent function scope.

The closure has access to variables in three scopes:

* Variables declared in their own scope
* Variables declared in a parent function scope
* Variables declared in the global namespace

**var** globalVar = "abc";

*// Parent self invoking function*

(**function** **outerFunction** (outerArg) { *// begin of scope outerFunction*

*// Variable declared in outerFunction function scope*

**var** outerFuncVar = 'x';

*// Closure self-invoking function*

(**function** **innerFunction** (innerArg) { *// begin of scope innerFunction*

*// variable declared in innerFunction function scope*

**var** innerFuncVar = "y";

console.log(

"outerArg = " + outerArg + "\n" +

"outerFuncVar = " + outerFuncVar + "\n" +

"innerArg = " + innerArg + "\n" +

"innerFuncVar = " + innerFuncVar + "\n" +

"globalVar = " + globalVar);

}*// end of scope innerFunction)(5); // Pass 5 as parameter*

}*// end of scope outerFunction )(7); // Pass 7 as parameter*

innerFunction is closure that is defined inside outerFunction and has access to all variables declared and defined in the outerFunction scope. In addition, the function defined inside another function as a closure will have access to variables declared in the global namespace.

Thus, the output of the code above would be:

outerArg = 7

outerFuncVar = x

innerArg = 5

innerFuncVar = y

globalVar = abc

## Question 5

### Write a mul function which will produce the following outputs when invoked:

console.log(mul(2)(3)(4)); *// output : 24*

console.log(mul(4)(3)(4)); *// output : 48*

Below is the answer followed by an explanation to how it works:

**function** **mul** (x) {

**return** **function** (y) { *// anonymous function*

**return** **function** (z) { *// anonymous function*

**return** x \* y \* z;

};

};

}

Here the mul function accepts the first argument and returns an anonymous function, which takes the second parameter and returns another anonymous function that will take the third parameter and return the multiplication of the arguments that have been passed.

In JavaScript, a function defined inside another one has access to the outer function's variables. Therefore, a function is a first-class object that can be returned by other functions as well and be passed as an argument in another function.

* A function is an instance of the Object type
* A function can have properties and has a link back to its constructor method
* A function can be stored as a variable
* A function can be pass as a parameter to another function
* A function can be returned from another function

## Question 6

### How to empty an array in JavaScript?

For instance,

**var** arrayList = ['a','b','c','d','e','f'];

**How can we empty the array above?**

There are a couple ways we can use to empty an array, so let's discuss them all.

#### Method 1

arrayList = []

Above code will set the variable arrayList to a new empty array. This is recommended if you don't have **references to the original array** arrayList anywhere else, because it will actually create a new, empty array. You should be careful with this method of emptying the array, because if you have referenced this array from another variable, then the original reference array will remain unchanged.

For Instance,

**var** arrayList = ['a','b','c','d','e','f']; *// Created array*

**var** anotherArrayList = arrayList; *// Referenced arrayList by another variable*

arrayList = []; *// Empty the array*

console.log(anotherArrayList); *// Output ['a','b','c','d','e','f']*

#### Method 2

arrayList.length = 0;

The code above will clear the existing array by setting its length to 0. This way of emptying the array also updates all the reference variables that point to the original array. Therefore, this method is useful when you want to update all reference variables pointing to arrayList.

For Instance,

**var** arrayList = ['a','b','c','d','e','f']; *// Created array*

**var** anotherArrayList = arrayList; *// Referenced arrayList by another variable*

arrayList.length = 0; *// Empty the array by setting length to 0*

console.log(anotherArrayList); *// Output []*

#### Method 3

arrayList.splice(0, arrayList.length);

The implementation above will also work perfectly. This way of emptying the array will also update all the references to the original array.

**var** arrayList = ['a','b','c','d','e','f']; *// Created array*

**var** anotherArrayList = arrayList; *// Referenced arrayList by another variable*

arrayList.splice(0, arrayList.length); *// Empty the array by setting length to 0*

console.log(anotherArrayList); *// Output []*

#### Method 4

**while**(arrayList.length){

arrayList.pop();

}

The implementation above can also empty arrays, but it is usually not recommended to use this method often.

## Question 7

### How do you check if an object is an array or not?

The best way to find out whether or not an object is an instance of a particular class is to use the toString method from Object.prototype:

**var** arrayList = [1,2,3];

One of the best use cases of type-checking an object is when we do method overloading in JavaScript. For example, let's say we have a method called greet, which takes one single string and also a list of strings. To make our greet method workable in both situations, we need to know what kind of parameter is being passed. Is it a single value or a list of values?

**function** **greet**(param){

**if**(){ *// here have to check whether param is array or not*

}**else**{

}

}

However, as the implementation above might not necessarily check the type for arrays, we can check for a single value string and put some array logic code in the else block. For example:

**function** **greet**(param){

**if**(**typeof** param === 'string'){

}**else**{

*// If param is of type array then this block of code would execute*

}

}

Now it's fine we can go with either of the aforementioned two implementations, but when we have a situation where the parameter can be single value, array, and object type, we will be in trouble.

Coming back to checking the type of an object, as mentioned previously we can use  
Object.prototype.toString

**if**( Object.prototype.toString.call( arrayList ) === '[object Array]' ) {

console.log('Array!');

}

If you are using jQuery, then you can also use the jQuery isArray method:

**if**($.isArray(arrayList)){

console.log('Array');

}**else**{

console.log('Not an array');

}

FYI, jQuery uses Object.prototype.toString.call internally to check whether an object is an array or not.

In modern browsers, you can also use

Array.isArray(arrayList);

Array.isArray is supported by Chrome 5, Firefox 4.0, IE 9, Opera 10.5 and Safari 5

## Question 8

### What will be the output of the following code?

**var** output = (**function**(x){

**delete** x;

**return** x;

})(0);

console.log(output);

The output would be 0. The delete operator is used to delete properties from an object. Here x is not an object but a **local variable**. delete operators don't affect local variables.

## Question 9

### What will be the output of the following code?

**var** x = 1;

**var** output = (**function**(){

**delete** x;

**return** x;

})();

console.log(output);

The output would be 1. The delete operator is used to delete the property of an object. Here x is not an object, but rather it's the **global variable** of type number.

## Question 10

### What will be the output of the code below?

**var** x = { foo : 1};

**var** output = (**function**(){

**delete** x.foo;

**return** x.foo;

})();

console.log(output);

The output would be undefined. The delete operator is used to delete the property of an object. Here, x is an object which has the property foo, and as it is a self-invoking function, we will delete the foo property from object x. After doing so, when we try to reference a deleted property foo, the result isundefined.

## Question 11

### What will be the output of the code below?

**var** Employee = {

company: 'xyz'

}

**var** emp1 = Object.create(Employee);

**delete** emp1.company

console.log(emp1.company);

The output would be xyz. Here, emp1 object has company as its **prototype** property. The delete operator doesn't delete prototype property.

emp1 object doesn't have **company** as its own property. You can test it console.log(emp1.hasOwnProperty('company')); //output : false. However, we can delete the company property directly from theEmployee object using delete Employee.company. Or, we can also delete the emp1 object using the \_\_proto\_\_ property delete emp1.\_\_proto\_\_.company.

## Question 12

### What is undefined x 1 in JavaScript?

**var** trees = ["redwood","bay","cedar","oak","maple"];

**delete** trees[3];

When you run the code above and type console.log(trees); into your Chrome developer console, you will get  
["redwood", "bay", "cedar", undefined × 1, "maple"]. When you run the code in Firefox's browser console, you will get ["redwood", "bay", "cedar", undefined, "maple"]. Thus, it's clear that the Chrome browser has its own way of displaying uninitialised indexes in arrays. However, when you check trees[3] === undefined in both browsers, you will get similar output as true.

**Note:** Please remember you do not need to check for the uninitialised index of array in trees[3] === 'undefined × 1', as it will give you an error. 'undefined × 1' is just way of displaying an array's uninitialised index in Chrome.

## Question 13

### What will be the output of the code below?

**var** trees = ["xyz","xxxx","test","ryan","apple"];

**delete** trees[3];

console.log(trees.length);

The output would be 5. When we use the delete operator to delete an array element, the array length is not affected from this. This holds even if you deleted all elements of an array using the delete operator.

In other words, when the delete operator removes an array element, that deleted element is not longer present in array. In place of value at deleted index undefined x 1 in **chrome** and undefined is placed at the index. If you do console.log(trees) output ["xyz", "xxxx", "test", undefined × 1, "apple"] in Chrome and in Firefox ["xyz", "xxxx", "test", undefined, "apple"].

## Question 14

### What will be the output of the code below?

**var** bar = true;

console.log(bar + 0);

console.log(bar + "xyz");

console.log(bar + true);

console.log(bar + false);

The code will output 1, "truexyz", 2, 1. Here's a general guideline for addition operators:

* Number + Number -> Addition
* Boolean + Number -> Addition
* Boolean + Number -> Addition
* Number + String -> Concatenation
* String + Boolean -> Concatenation
* String + String -> Concatenation

## Question 15

### What will be the output of the code below?

**var** z = 1, y = z = **typeof** y;

console.log(y);

The output would be undefined. According to the associativity rule, operators with the same precedence are processed based on the associativity property of the operator. Here, the associativity of the assignment operator is Right to Left, so typeof y will evaluate first , which is undefined. It will be assigned to z, and then y would be assigned the value of z and then z would be assigned the value 1.

## Question 16

### What will be the output of the code below?

*// NFE (Named Function Expression*

**var** foo = **function** **bar**(){ **return** 12; };

**typeof** bar();

The output would be Reference Error. To make the code above work, you can re-write it as follows:

**Sample 1**

**var** bar = **function**(){ **return** 12; };

**typeof** bar();

or

**Sample 2**

**function** **bar**(){ **return** 12; };

**typeof** bar();

A function definition can have only one reference variable as its function name. In **sample 1**, bar's reference variable points to anonymous function. In **sample 2**, the function's definition is the name function.

**var** foo = **function** **bar**(){

*// foo is visible here*

*// bar is visible here*

console.log(**typeof** bar()); *// Work here :)*

};

*// foo is visible here*

*// bar is undefined here*

## Question 17

### What is the difference between the function declarations below?

**var** foo = **function**(){

*// Some code*

};

**function** **bar**(){

*// Some code*

};

The main difference is the function foo is defined at run-time whereas function bar is defined at parse time. To understand this in better way, let's take a look at the code below:

Run-Time **function** **declaration**

<**script**>

**foo**(); // **Calling** **foo** **function** **here** **will** **give** **an** **Error**

**var** **foo** = **function**(){

console.log("Hi I am inside Foo");

};

</script>

<script>

Parse-Time **function** **declaration**

**bar**(); // **Calling** **foo** **function** **will** **not** **give** **an** **Error**

**function** **bar**(){

console.log("Hi I am inside Foo");

};

</script>

Another advantage of this first-one way of declaration is that you can declare functions based on certain conditions. For example:

<script>

**if**(testCondition) {*// If testCondition is true then*

**var** foo = **function**(){

console.log("inside Foo with testCondition True value");

};

}**else**{

**var** foo = **function**(){

console.log("inside Foo with testCondition false value");

};

}

</script>

However, if you try to run similar code using the format below, you'd get an error:

<script>

**if**(testCondition) {*// If testCondition is true then*

**function** **foo**(){

console.log("inside Foo with testCondition True value");

};

}**else**{

**function** **foo**(){

console.log("inside Foo with testCondition false value");

};

}

</script>

## Question 18

### What is function hoisting in JavaScript?

**Function Expression**

**var** foo = **function** **foo**(){

**return** 12;

};

In JavaScript, variable and functions are hoisted. Let's take function hoisting first. Basically, the JavaScript interpreter looks ahead to find all variable declarations and then hoists them to the top of the function where they're declared. For example:

foo(); *// Here foo is still undefined*

**var** foo = **function** **foo**(){

**return** 12;

};

Behind the scene of the code above looks like this:

**var** foo = undefined;

foo(); *// Here foo is undefined*

foo = **function** **foo**(){

/ Some code stuff

}

**var** foo = undefined;

foo = **function** **foo**(){

/ Some code stuff

}

foo(); *// Now foo is defined here*

## Question 19

### What will be the output of code below?

**var** salary = "1000$";

(**function** () {

console.log("Original salary was " + salary);

**var** salary = "5000$";

console.log("My New Salary " + salary);

})();

The output would be undefined, 5000$. Newbies often get tricked by JavaScript's hoisting concept. In the code above, you might be expecting salary to retain its value from the outer scope until the point that salary gets re-declared in the inner scope. However, due to hoisting, the salary value was undefined instead. To understand this better, have a look of the code below:

**var** salary = "1000$";

(**function** () {

**var** salary = undefined;

console.log("Original salary was " + salary);

salary = "5000$";

console.log("My New Salary " + salary);

})();

salary variable is hoisted and declared at the top in the function's scope. The console.log inside returns undefined. After the console.log, salary is redeclared and assigned 5000$.

## Question 20

### What is the instanceof operator in JavaScript? What would be the output of the code below?

**function** **foo**(){

**return** foo;

}

**new** foo() **instanceof** foo;

Here, instanceof operator checks the current object and returns true if the object is of the specified type.

For Example:

**var** dog = **new** Animal();

dog **instanceof** Animal *// Output : true*

Here dog instanceof Animal is true since dog inherits from Animal.prototype.

**var** name = **new** String("xyz");

name **instanceof** String *// Output : true*

Here name instanceof String is true since dog inherits from String.prototype. Now let's understand the code below:

**function** **foo**(){

**return** foo;

}

**new** foo() **instanceof** foo;

Here function foo is returning foo, which again points to function foo.

**function** **foo**(){

**return** foo;

}

**var** bar = **new** foo();

*// here bar is pointer to function foo(){return foo}.*

So the new foo() instanceof foo return false;

[Ref Link](http://stackoverflow.com/questions/2449254/what-is-the-instanceof-operator-in-javascript)

## Question 21

### If we have a JavaScript associative array

**var** counterArray = {

A : 3,

B : 4

};

counterArray["C"] = 1;

### How can we calculate the length of the above associative array's counterArray?

There are no in-built functions and properties available to calculate the length of associative array object here. However, there are other ways by which we can calculate the length of an associative array object. In addition to this, we can also extend an Object by adding a method or property to the prototype in order to calculate length. However, extending an object might break enumeration in various libraries or might create cross-browser issues, so it's not recommended unless it's necessary. Again, there are various ways by which we can calculate length.

Object has the keys method which can be used to calculate the length of an object:

We can also calculate the length of an object by iterating through an object and by counting the object's own property.

```javascript

function getSize(object){

var count = 0;

for(key in object){

// hasOwnProperty method check own property of object

if(object.hasOwnProperty(key)) count++;

}

return count;

}

We can also add a length method directly on Object:

Object.length = **function**(){

**var** count = 0;

**for**(key **in** object){

*// hasOwnProperty method check own property of object*

**if**(object.hasOwnProperty(key)) count++;

}

**return** count;

}

*//Get the size of any object using*

console.log(Object.length(counterArray))

**Bonus**: We can also use Underscore (recommended, As it's lightweight) to calculate object length.