**1. ReactJS - Introduction:**

ReactJS is a JavaScript library for building user interfaces. It allows you to create reusable UI components and efficiently update and render them based on the application state.

**2. ReactJS - Understanding SPAs and MPAs:**

Single-Page Applications (SPAs) are web applications that load a single HTML page and dynamically update the content without requiring a full page reload. Multi-Page Applications (MPAs) consist of multiple HTML pages that are loaded separately.

**3. ReactJS - Creating our First React App:**

To create a React app, you can use the create-react-app (CRA) tool. It sets up a basic React project structure with the necessary configuration files. Here's an example of creating a new React app using CRA:

*// type this code on terminal window*

npx create-react-app my-app

cd my-app

npm start

**4. ReactJS - Project Structure created by CRA**:

The project structure created by CRA follows a convention. It includes folders such as `src` (source code), `public` (static assets), and files like `index.js` (entry point), `App.js` (main component), and `index.html` (HTML template).

**5. ReactJS - Introduction to JSX:**

JSX (JavaScript XML) is a syntax extension for JavaScript that allows you to write HTML-like code within JavaScript. It helps in creating React elements and describing the structure of the UI. JSX code is transformed into regular JavaScript code during the build process.

Example:

*//jsx code*

const element = <h1>Hello, JSX!</h1>;

**6. ReactJS - JSX Behind the Scenes:**

JSX is transformed into React.createElement() calls, which creates React elements. React elements are JavaScript objects that describe what should be rendered on the screen.

Example:

*//jsx code*

const element = React.createElement('h1', null, 'Hello, JSX!');

**7. ReactJS - Adding Styles to React Elements:**

You can add styles to React elements using inline styles or by importing CSS files. Inline styles use JavaScript objects to define styles, while CSS files can be imported and applied to components.

Example using inline styles:

*//jsx code*

const element = <h1 style={{ color: 'red', fontSize: '24px' }}>Styled Heading</h1>;

**8. ReactJS - Create React Elements Dynamically:**

React elements can be created dynamically using JavaScript expressions. This allows you to generate elements based on variables or data.

Example:

*//jsx code*

const name = 'John Doe';

const element = <h1>Hello, {name}!</h1>;

**9. ReactJS - Creating our First React Component:**

React components are the building blocks of a React application. They can be created as functions or classes. Here's an example of creating a functional component:

*//jsx code*

function Greeting() {

return <h1>Hello, React Component!</h1>;

}

**10. ReactJS - Passing Data to Components using Props:**

Props (short for properties) are used to pass data from a parent component to a child component. Props are passed as attributes to the child component in JSX.

Example:

*//jsx code*

function Greeting(props) {

return <h1>Hello, {props.name}!</h1>;

}

*// Usage:*

<Greeting name="John" />

**11. ReactJS - Import and Export of Modules:**

In React, you can import and export modules to organize your code and reuse components across different files. The `export` keyword is used to export a module, and the `import` keyword is used to import modules.

**Example of exporting a module:**

*//jsx code*

*// MyComponent.js*

import React from 'react';

function MyComponent() {

// Component code...

}

export default MyComponent;

**Example of importing a module:**

*//jsx code*

*// App.js*

import React from 'react';

import MyComponent from './MyComponent';

function App() {

return <MyComponent />;

}

**12. ReactJS - Introduction to CSS Modules:**

CSS Modules is a feature in React that allows you to write modular and scoped CSS styles for your components. It prevents CSS class name conflicts and encapsulates styles within the component.

Example:

*//jsx code*

*// styles.module.css*

.container {

background-color: #f0f0f0;

padding: 10px;

}

*// MyComponent.js*

import React from 'react';

import styles from './styles.module.css';

function MyComponent() {

return <div className={styles.container}>Component content</div>;

}

**13. ReactJS - Creating Mobile Responsive Components:**

React provides several techniques for creating mobile responsive components. CSS media queries, CSS frameworks like Bootstrap, or React libraries like React Responsive can be used to achieve responsiveness.

Example using CSS media queries:

*//jsx code*

*// MyComponent.css*

.container {

width: 100%;

}

@media (min-width: 768px) {

.container {

width: 50%;

}

}

*// MyComponent.js*

import React from 'react';

import './MyComponent.css';

function MyComponent() {

return <div className="container">Component content</div>;

}

**14. ReactJS - Stateful vs Stateless Components:**

Stateful components (also known as class-based components) have internal state and lifecycle methods, while stateless components (also known as functional components) are primarily responsible for rendering UI based on the props they receive.

Example of a stateful component:

*//jsx code*

class Counter extends React.Component {

constructor(props) {

super(props);

this.state = { count: 0 };

}

increment() {

this.setState({ count: this.state.count + 1 });

}

render() {

return (

<div>

<p>Count: {this.state.count}</p>

<button onClick={() => this.increment()}>Increment</button>

</div>

);

}

}

**15. ReactJS - Creating Class-based Components:**

Class-based components are created by extending the `React.Component` class and implementing the `render()` method. They can have state, lifecycle methods, and event handlers.

Example:

*//jsx code*

class MyComponent extends React.Component {

render() {

return <h1>Hello, Class-based Component!</h1>;

}

}

**16. ReactJS - More about setState() Method:**

The `setState()` method is used to update the state of a component. It accepts an object that represents the new state or a function that returns the new state based on the previous state. It also provides a callback that is executed after the state update is applied.

Example:

*//jsx code*

class Counter extends React.Component {

constructor(props) {

super(props);

this.state = { count: 0 };

}

increment() {

this.setState((prevState) => ({ count: prevState.count + 1 }));

}

render() {

return (

<div>

<p>Count: {this.state.count}</p>

<button onClick={() => this.increment()}>Increment</button>

</div>

);

}

}

**17. ReactJS - Passing Props to Class-based Components:**

Props can be passed to class-based components in the same way as functional components. They are accessible within the component via `this.props`.

Example:

*//jsx code*

class Greeting extends React.Component {

render() {

return <h1>Hello, {this.props.name}!</h1>;

}

}

*// Usage:*

<Greeting name="John" />

**18. ReactJS - Passing Function as Props:**

Functions can be passed as props to components, allowing child components to communicate with parent components. The parent component can pass a function as a prop, and the child component can invoke it when needed.

Example:

*//jsx code*

class ParentComponent extends React.Component {

handleClick() {

console.log('Button clicked!');

}

render() {

return <ChildComponent handleClick={this.handleClick} />;

}

}

class ChildComponent extends React.Component {

render() {

return <button onClick={this.props.handleClick}>Click me</button>;

}

}

**19. ReactJS - Practice Problem: Product Details Page:**

For this practice problem, let's create a product details page component that displays information about a product. Here's an example implementation:

*//jsx code*

import React from 'react';

function ProductDetailsPage(props) {

return (

<div>

<h1>{props.product.name}</h1>

<p>{props.product.description}</p>

<p>Price: {props.product.price}</p>

</div>

);

}

*// Usage:*

const product = {

name: 'Product Name',

description: 'Product Description',

price: 19.99,

};

<ProductDetailsPage product={product} />;

**20. ReactJS - Practice Problem: Product Details Page [Solution]:**

Solution to the previous practice problem that includes additional features such as conditional rendering and styling:

*//jsx code*

import React from 'react';

import './ProductDetailsPage.css';

function ProductDetailsPage(props) {

const { product } = props;

return (

<div className="product-details">

<h1 className="product-name">{product.name}</h1>

{product.description && <p className="product-description">{product.description}</p>}

<p className="product-price">Price: {product.price}</p>

{product.inStock ? (

<button className="add-to-cart">Add to Cart</button>

) : (

<p className="out-of-stock">Out of Stock</p>

)}

</div>

);

}

*// Usage:*

const product = {

name: 'Product Name',

description: 'Product Description',

price: 19.99,

inStock: true,

};

<ProductDetailsPage product={product} />;

**21. ReactJS - render() method - Behind the scenes:**

The `render()` method in React is responsible for rendering the component's UI. It returns a React element or `null` (for no rendering). React determines the difference between the previous and current rendered elements and updates the DOM efficiently.

**22. ReactJS - Component Lifecycle - Creation:**

React components have a lifecycle that consists of several methods called at different stages of a component's existence. The creation lifecycle methods are `constructor()`, `componentDidMount()`, and `render()`.

Example:

*//jsx code*

class MyComponent extends React.Component {

constructor(props) {

super(props);

*// Initialization code...*

}

componentDidMount() {

*// Code to run after the component is mounted...*

}

render() {

*// Rendering code...*

}

}

**23. ReactJS - Component Lifecycle - Updation:**

The updation lifecycle methods in React are called when a component's props or state change. The methods include `shouldComponentUpdate()`, `componentDidUpdate()`, and `render()`.

Example:

*//jsx code*

class MyComponent extends React.Component {

shouldComponentUpdate(nextProps, nextState) {

*// Return true if the component should update, false otherwise...*

}

componentDidUpdate(prevProps, prevState) {

*// Code to run after the component is updated...*

}

render() {

*// Rendering code...*

}

}

**24. ReactJS - shouldComponentUpdate() Lifecycle Method Explain In detailed coding examples in React JS:**

The `shouldComponentUpdate()` method is used to optimize performance by controlling whether a component should update or not. It receives the next props and state as arguments and returns a boolean value indicating whether the component should update.

Example:

*//jsx code*

class MyComponent extends React.Component {

shouldComponentUpdate(nextProps, nextState) {

*// Perform custom logic to determine if the component should update...*

if (nextProps.someProp === this.props.someProp) {

*// If the prop hasn't changed, do not update*

return false;

}

return true;

}

render() {

*// Rendering code...*

}

}