**Building Blocks of React.JS**

* Core Building Blocks of React

**These are the key building blocks of React:**

**Components:** React enables you to decompose your app's visual tree into smaller reusable units called components.

**JSX:** React's response to templates. JSX enables you to write components' template in code in a productive manner.

**Props and state:** Props enable parent-child component interaction, while state is used by components to manage its behaviour.

**Conditional rendering:** It is often needed in our application to show or hide various UI elements based on the state of the application.

* Components

Components are independent and reusable bits of code. They serve the same purpose as JavaScript functions, but work in isolation and return HTML.

Components come in two types, Class components and Function components, in this tutorial we will concentrate on Function components.

Example

Create a Class component called Car

class Car extends React.Component {

render() {

return <h2>Hi, I am a Car!</h2>;

}

}

Example

Create a Function component called Car

function Car() {

return <h2>Hi, I am a Car!</h2>;

}

* State and Props

**State**: React components has a built-in state object.

The state object is where you store property values that belongs to the component.

When the state object changes, the component re-renders.

**Example:**

class Car extends React.Component {

constructor(props) {

super(props);

this.state = {brand: "Ford"};

}

render() {

return (

<div>

<h1>My Car</h1>

</div>

);

}

}

**Props:** Props are arguments passed into React components. Props are passed to components via HTML attributes. props stands for properties.

function Car(props) {

return <h2>I am a { props.brand }!</h2>;

}

function Garage() {

return (

<>

<h1>Who lives in my garage?</h1>

<Car brand="Ford" />

</>);}

* Rendering

React renders HTML to the web page by using a function called render(). The purpose of the function is to display the specified HTML code inside the specified HTML element. In the render() method, we can read props and state and return our JSX code to the root component of our app. In the render() method, we cannot change the state, and we cannot cause side effects( such as making an HTTP request to the webserver).

Now your React application has a component called Car, which returns an <h2> element.

To use this component in your application, use similar syntax as normal HTML: <Car />

**Example**

Display the Car component in the "root" element:

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Car />);

* Conditional Rendering

We can create multiple components which encapsulate behaviour that we need. After that, we can render them depending on some conditions or the state of our application.  There is more than one way to do conditional rendering in React. They are given below.

* if
* ternary operator
* logical && operator
* switch case operator
* Conditional Rendering with enums
* What is JSX

The render function specifies the HTML output of a React component. JSX (JavaScript Extension), is a React extension which allows writing JavaScript code that looks like HTML.

JSX provides you to write HTML/XML-like structures (e.g., DOM-like tree structures) in the same file where you write JavaScript code, then pre-processor will transform these expressions into actual JavaScript code. Just like XML/HTML, JSX tags have a tag name, attributes, and children.

Example:

const myElement = <h1>I Love JSX!</h1>;

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(myElement);

* Example

import React, { Component } from 'react';

class App extends Component{

   render(){

      return(

         <div>

            <h1>JavaTpoint</h1>

          <h2>Training Institutes</h2>

            <p>This website contains the best CS tutorials.</p>

         </div>

      );

   }

}

export default App;

**Output:**

JavaTpoint

Training Institutes

This website contains the best CS tutorials.

* Why JSX
* It is faster than regular JavaScript because it performs optimization while translating the code to JavaScript.
* Instead of separating technologies by putting mark-up and logic in separate files, React uses components that contain both. We will learn components in a further section.
* It is type-safe, and most of the errors can be found at compilation time.
* It makes easier to create templates.
* Advantages of JSX
* JSX helps us in keeping our code simpler and elegant when writing large pieces of code.
* According to the React docs, most people find it helpful as a visual aid when working with UI inside the JavaScript code.
* JSX also allows React to show more useful error and warning messages.
* If one is familiar with HTML, it is quite easy to use JSX when building React application
* Faster than normal JavaScript as it performs optimizations while translating to regular JavaScript.
* Expressions in JSX

With JSX you can write expressions inside curly braces { }.

The expression can be a React variable, or property, or any other valid JavaScript expression. JSX will execute the expression and return the result:

Example

Execute the expression 5 + 5:

const myElement = <h1>React is {5 + 5} times better with JSX</h1>;

* Internal Implementation of JSX

Fundamentally the above JSX code gets compiled as:-

"use strict";

var element = React.createElement(

"div",

{ "class": "wrapper" },

" ",

React.createElement(

"h1",

{ color: "blue", className: "sidebar" },

"Hello, world!"

),

React.createElement(

"h2",

{ color: "blue", className: "sidebar" },

"Hey Guys"

)

);

Babel converts the JSX to React.CreateElement code .Babel Converts Jsx Code in to a Javascript code that the browser understands.Look at the example below.

**Inline Styles in JSX**:- With inline styles, you have to option to combine CSS syntax with JSX code.

var myStyle= {

color:'red',

backgroundColor:'black',

fontWeight:'bold'

};

var element= <div style={myStyle}>Hello World</div>;

* Hands-on: Creating a react component with jsx

import React from 'react';

import { makeStyles } from '@material-ui/core/styles';

import List from '@material-ui/core/List';

import ListItem from '@material-ui/core/ListItem';

import ListItemText from '@material-ui/core/ListItemText';

import ListSubheader from '@material-ui/core/ListSubheader';

const useStyles = makeStyles((theme) => ({

root: {

width: '100%',

maxWidth: 360,

backgroundColor: theme.palette.background.paper,

position: 'relative',

overflow: 'auto',

maxHeight: 300,

},

listSection: {

backgroundColor: 'inherit',

},

ul: {

backgroundColor: 'inherit',

padding: 0,

},

}));

export default function PinnedSubheaderList() {

const classes = useStyles();

return (

<List className={classes.root} subheader={<li />}>

{[0, 1, 2, 3, 4].map((sectionId) => (

<li key={`section-${sectionId}`} className={classes.listSection}>

<ul className={classes.ul}>

<ListSubheader>{`I'm sticky ${sectionId}`}</ListSubheader>

{[0, 1, 2].map((item) => (

<ListItem key={`item-${sectionId}-${item}`}>

<ListItemText primary={`Item ${item}`} />

</ListItem>

))}

</ul>

</li>

))}

</List>

);

}

**Output:**

