**What are components?**

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

1. Functional components
2. Class Components

**Functional components**

These components are purely presentational and are simply represented by a function that optionally takes props and returns a React element to be rendered to the page.

Generally, it is preferred to use functional components whenever possible because of their predictability and conciseness. Since, they are purely presentational, their output is always the same given the same props.

You may find functional components referred to as stateless, dumb or presentational in other literature. All these names are derived from the simple nature that functional components take on.

=> **Functional** because they are basically functions

=> **Stateless** because they do not hold and/or manage state

=> **Presentational** because all they do is output UI elements

A functional component in it’s simplest form looks something like this:

const Greeting = () => <h1>Hi, I’m a dumb component!</h1>;

const Greeting = props => <h1>Hello {props.name}</h1>;

**Class Components**

These components are created using ES6’s class syntax. They have some additional features such as the ability to contain logic (for example methods that handle onClick events), local state (more on this in the next chapter) and other capabilities to be explored in later sections of the book.

As you explore other resources, you might find class components referred to as smart, container or stateful components.

=> **Class** because they are basically classes

=> **Smart** because they can contain logic

=> **Stateful** because they can hold and/or manage local state

=> **Container** because they usually hold/contain numerous other (mostly functional) components

Class components have a considerably larger amount of markup. Using them excessively and unnecessarily can negatively affect performance as well as code readability, maintainability and testability.

A class component in its simplest form:

class Greeting extends React.Component {

render(){

return <h1>Hi, I’m a smart component!</h1>;

}

}

class Greeting extends React.Component {

render(){

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

}

}

**Installing the React Developer tools**

1. [Mozilla Firefox Add-ons](https://addons.mozilla.org/en-US/firefox/addon/react-devtools/)
2. [Chrome Web Store](https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi)

**State**

A state is a variable which exists inside a component, that cannot be accessed and modified outside the component and can only be used inside the component. Works very similarly to a variable that is declared inside a function that cannot be accessed outside the scope of the function in normal JavaScript. State Can be modified using this.setState. State can be asynchronous. Whenever this.setState is used to change the state class is render itself.

**State in class componets**

import React from 'react'

class MyComponent extends React.Component {

  constructor(props){

    super(props);

    this.state = { date: new Date(), name: 'Kofi'};

  }

  render(){

    return(

      <div>

            <p> Hello {this.state.name} , it is {this.state.toLocaleTimeString()

            <p>Date: {this.state.date.toLocaleDateString()}

      </div>

    )

  }

}

**State in Functional Components (Hooks)**

import React, {useState} from 'react';

function MyComponent(){

  const [date, setDate] = useState(new Date())

  const [name, setName] = useState("Kofi");

  return(

      <div>

            <p> Hello {date.name} , it is {date.toLocaleTimeString()

            <p>Date: {date.toLocaleDateString()}

            <button onClick={setDate(new Date())}></button>

      </div>

  )

}

**initialization**

**mounting**

**updating, and**

**unmounting.**

**Initialization**

This is the phase in which the component is going to start its journey by setting up the state (see below) and the props. This is usually done inside the constructor method (see below to understand the initialization phase better).

class Initialize extends React.Component {

    constructor(props)

    {

    // Calling the constructor of

    // Parent Class React.Component

    super(props);

    // initialization process

    this.state = {

       date : new Date(),

       clickedStatus: false

     };

}

**Mounting**

The name is self-explanatory. Mounting is the phase in which our React component mounts on the DOM (i.e., is created and inserted into the DOM).

This phase comes onto the scene after the initialization phase is completed. In this phase, our component renders the first time. The methods that are available in this phase are:

**1. componentWillMount()**

This method is called just before a component mounts on the DOM or the render method is called. After this method, the component gets mounted.

Note: You should not make API calls or any data changes using this.setstate in this method because it is called before the render method. So, nothing can be done with the DOM (i.e. updating the data with API response) as it has not been mounted. Hence, we can’t update the state with the API response.

**2. componentDidMount()**

This method is called after the component gets mounted on the DOM. Like componentWillMount, it is called once in a lifecycle. Before the execution of this method, the render method is called (i.e., we can access the DOM). We can make API calls and update the state with the API response**.**

**Have a look to understand these mounting methods:**

class LifeCycle extends React.Component {

    componentWillMount() {

        console.log('Component will mount!')

     }

    componentDidMount() {

        console.log('Component did mount!')

        this.getList();

     }

    getList=()=>{

     /\*\*\* method to make api call\*\*\*

    }

    render() {

        return (

           <div>

              <h3>Hello mounting methods!</h3>

           </div>

        );

     }

  }

**Updating**

This is the third phase through which our component passes. After the mounting phase where the component has been created, the update phase comes into the scene. This is where component’s state changes and hence, re-rendering takes place.

In this phase, the data of the component (state & props) updates in response to user events like clicking, typing and so on. This results in the re-rendering of the component. The methods that are available in this phase are:

**shouldComponentUpdate()**

This method determines whether the component should be updated or not. By default, it returns true. But at some point, if you want to re-render the component on some condition, then shouldComponentUpdate method is the right place**.**

Suppose, for example, you want to only re-render your component when there is a change in prop — then utilize the power of this method. It receives arguments like nextProps and nextState which help us decide whether to re-render by doing a comparison with the current prop value.

**2. componentWillUpdate()**

Like other methods, its name is also self-explanatory. It is called before the re-rendering of the component takes place. It is called once after the ‘shouldComponentUpdate’ method. If you want to perform some calculation before re-rendering of the component and after updating the state and prop, then this is the best place to do it. Like the ‘shouldComponentUpdate’ method, it also receives arguments like nextProps and nextState.

**3. ComponentDidUpdate()**

This method is called just after the re-rendering of the component. After the new (updated) component gets updated on the DOM, the ‘componentDidUpdate’ method is executed. This method receives arguments like prevProps and prevState.

**Have a look to understand the updating methods better:**

class LifeCycle extends React.Component {

      constructor(props)

      {

        super(props);

         this.state = {

           date : new Date(),

           clickedStatus: false,

           list:[]

         };

      }

      componentWillMount() {

          console.log('Component will mount!')

       }

      componentDidMount() {

          console.log('Component did mount!')

          this.getList();

       }

      getList=()=>{

       /\*\*\* method to make api call\*\*\*

       fetch('https://api.mydomain.com')

          .then(response => response.json())

          .then(data => this.setState({ list:data }));

      }

       shouldComponentUpdate(nextProps, nextState){

         return this.state.list!==nextState.list

        }

       componentWillUpdate(nextProps, nextState) {

          console.log('Component will update!');

       }

       componentDidUpdate(prevProps, prevState) {

          console.log('Component did update!')

       }

      render() {

          return (

             <div>

                <h3>Hello Mounting Lifecycle Methods!</h3>

             </div>

          );

       }

}

**Unmounting**

This is the last phase in the component’s lifecycle. As the name clearly suggests, the component gets unmounted from the DOM in this phase. The method that is available in this phase is:

**1. componentWillUnmount()**

This method is called before the unmounting of the component takes place. Before the removal of the component from the DOM, ‘componentWillUnMount’ executes. This method denotes the end of the component’s lifecycle.