11. React States

11.1 Definition of state: what is the difference between state and props?

- **Props (quick revision)**: Props stand for properties. They are JavaScript objects and are used to pass data from a parent component to a child component in React. In short, props are used to create communication between React components. You can place a piece of code in a function and reuse it over and over. We know that React components are JavaScript functions which return JSX. We also know that components are meant to be re-used. Think of props as parameters of a function (or parameter to your React component). Props are the values that are passed into components to be used. **Note**: Unlike parameters, props should not be changed. If any internal changes need to happen, use states.

- **States**: State is a built in JavaScript object that is used to contain data about a component's current situation. As stated above, React components are JavaScript functions which return JSX. You can understand states as local variables of a function which can only be accessed and changed within component. A state in React Component is its own local state; it cannot be accessed and modified outside the component.

    - **Why we need states**: State comes in handy when you want to update and re-render your component based on something the user has done. Good example is when we have form (input) and when the user updates the form with her new information, we will use state to store the user's updated value to manage the updated information. In short, state is used to keep track of information between multiple renderings of a component. **Example**: Tracking information about the component before and after clicking a button

    - **Why not just use variables than states in React**: Though using state may seem similar to class variable, the major difference between using class variables and state is updating data. When you use states, you do not have to manually reassigning the variable, you just use the this.setState() method to change value of states.

    - **Note**: A component itself can create, update and use state

    - **Note**: Just like any variable, a component's state can change over time. When the state object changes, the component re-renders.

- **Similarity between props and state**: Both are in object format in React containing key-value pairs that we use to render the value of the objects. Both hold information about a component
- **Difference between props and States**:
    - Props are variables that get passed to the component from parent component (like a parameter is passed to a function). State on the other hand is still variable, but directly initialized and managed within the component.
    - Props are objects of React with immutable properties, meaning, we cannot change their value throughout the component. Because props contain information set by the parent component, that information should not be changed. This is because changing value of props will not make sense, it will be like changing the parameter of an adder function to fixed parameter. States are also objects of React, however, states can be changed by the component they are initialized in, just like we can change value of a variable
        - **Why are props called pure?** Because props are passed in another component and they cannot change, any component that uses props only (no state) is pure, meaning, it will render the same output as long as we provide the same input
    - State is created in the component itself (every component has a built-in state), however, props are passed from a parent component to a child
    - Props are available in both classes based and function-based components without any issue. However, state used only available in class-based functions because it has constructor function where states can be initiated. States became available in functional component only after 2019 when Hooks were created.
    - Props come in handy when you want to show some static information inside of a component, meaning, information that won't need to be updated in the component. However, we use state when we want to update and re-render a component based on something a user has done. A good example is when we have form (input or checkbox) and a user updates the form with information, we will use state to store the user's updated value and to manage the updated information

11.2 Steps to add and use states in your class component

- **How to initialize and use states in React components**:
    - **Before 2019**: Up until the introduction of React Hooks in 2019, the way to use states was by using class components. We could not use functional components and take advantage of States as well. That was because the way we initialize states in a component was only inside a constructor function which only was possible if the component is Class based component.
    - **After 2019 (after Hooks)**: The introduction of Hooks lets us use states and other React features without writing a class. We will cover both ways of using states below
- **Steps to add and use states on your class components**: While using class components, we create states when we create our component. States can be initialized inside the constructor function of the component or directly without constructor function
    - **Step 1**: Create your class component
    - **Step 2**: Add the constructor function in your component. As a revision, the constructor is a method (built in function) in a class that's automatically called when we want to create an object instance from that class. When we use the constructor method in React class components, the constructor calls the parent class, called Component, and helps in constructing the state object from that class.
    - **Step 3**: In your constructor, call the "super()" function. This is to set a property or access 'this' inside the constructor in your component. In JavaScript, super() function calls the parent class constructor. If we create an ES6 class or a function, and extend another class from it, then the child class can access the methods/variables of the parent class using super() method. In ES6 class constructors MUST call super() if they are subclasses extending from a parent class.
    - **Step 4**: Add an object called "state" in your constructor. To do this, add "this.state" right after calling super() and set it equal to an empty object.
    - **Step 5**: Once we have created the empty object, you can fill it with data of whatever key and value pair you want.

- **Step 6**: To be able to access state from the render method, we just call the value of the state we want to use inside the JSX returned under the render method. You could do that in a paragraph tag like below:

```
<p>{this.state.topping}</p>
```

- **Using states in class components by initializing states in the constructor**

```
import React, { Component } from "react";
class ExampleComp extends Component {
constructor() {
super();
this.state = {
bussinessName: "EvangadiTech",
businessState: "Maryland",
Country: "USA",
};
}
render() {
return (
<div><h1>{this.state.bussinessName}</h1></div>
);
}
}
export default ExampleComp;
```

- **Using states in class components by initialization states without constructor**: Initialization without constructor is not widely adopted yet. This is because class property declarations are new features in JavaScript. But once they will be supported by all

browsers, they may be used more widely in React to create initial state in a class component. **Note**: If you do not use a constructor, you can use the state object directly, without the "this.state". You will need to use the "this.state" if you use a constructor because state is an object that belongs to the "ExampleComp" class which has a method called constructor and you are trying to access the state from inside of this constructor method. You can remove the constructor function altogether and initialize states as follows:

```
class ExampleComp extends Component {
    state = {
    bussinessName: "EvangadiTech",
    businessState: "Maryland",
    };
    render() {
    return (
    <div>
    <h1>{this.state.bussinessName}</h1>
    </div>
    );
    }
    }
    export default ExampleComp;
```

- **Passing state to external components as props**: We can also pass state values to child components as props. Below, the ExampleComp component is state and renders the OtherExample component as a child. Let us pass the "state1" property found in its state down to this child component then show the value of "state1" in OtherExample component's render method. The value "0" will be displayed in your browser/localhost

// ExampleComp.js (parent component)

```
import OtherExample from "./OtherExample";

class ExampleComp extends Component {

constructor() {

super();

this.state = {

state1: 0,

state2: "Test",

};

}

render() {

return (

<div>

<OtherExample myData={this.state.state2} />

</div>

);

}

}

export default ExampleComp;
```

// OtherExample.js (child component)

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

class OtherExample extends Component {

render() {

return <div>{this.props. myData}</div>;

}

}

export default OtherExample;
```

11.3 Updating state values in your class component: handling events in react

- Before we talk about updating values of a state, let us talk how to handle events in React
- **Event handlers**: Event handlers determine what action is to be taken whenever an event is fired. This could be a button click or a change in a text input. Essentially, event handlers are what make it possible for users to interact with your React app. In vanilla JavaScript, if you want to add an onclick event handler on a button element, you first need to select the DOM element, then you need to attach an event listener with the .addEventListener() method. Inside this method, you can call or define another function that will execute when the event is triggered.
- **Handling events in React**: The way you bind an event handler with a React element is very similar with how we did it when we bind JavaScript event handlers with HTML element, except few exceptions. In React, each component has its own markup and methods, and in most cases, there is no need to add or remove event listeners. You can simply declare the function to be called inside your component, and immediately call it inside the return() method. Below, let us compare event handling in vanilla JavaScript and React.
    - **Event handling in vanilla JS**: <button onclick="sayHello()"> Say Hello <button>
        - **Note**: Unlike vanilla JavaScript (DOM) events, React events are named using lowercase
        - **Note**: DOM event handlers appear as a string
        - **Note**: You simply return false to avoid default behavior in HTML
    - **Event handling in React**: <button onClick={sayHello}> Say Hello <button>

- **Note**: Unlike vanilla JavaScript events, React events are named using camelCase (example: onClick), rather than lowercase
- **Note**: React event handlers appear inside curly braces rather than as a string
- **Note**: You in React, you must explicitly call preventDefault to avoid default behavior
- **Note**: Make sure you are not calling the event handler function (make sure to call the function without brackets) when you pass it to the component

- **Steps to take when handling events in React**:
  - **Step 1**: Create a component
  - **Step 2**: Define a JavaScript function inside your class as a method.
    - **Binding required for class components**: In class components, if you want your function to access pass props or state from your component, you need to bind the function to your component. In JavaScript, class methods (functions in a class) are not bound to class instances by default. Therefore, you need to use the "this" key word to bind the function to the class instance (your component in your case). There are different ways of binding a class method to your class component
  - **Step 3**: Return an HTML element in your component and execute your event handler function when an event happens on the HTML element.

- **Handling events in class components**: Handling events when your event handler function does not need to access the components props and states

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

class ExampleComp extends Component {
```

```
sayHi() {
alert("hi");
}
render() {
return
<button onClick={this.sayHi}>
Click me </button>;
}
}
export default ExampleComp;
```

- **Handling events in class components:** Binding event handling function to your class component inside the constructor by calling the .bind(this)

```
import React, { Component } from "react";
class ExampleComp extends Component {
constructor(props) {
super(props);
this.sayHi = this.sayHi.bind(this);
}
sayHi() {
console.log(this); //"this" refers to ExampleComp.js
}
render() {
return <button onClick={this.sayHi}> ClickMe </button>;
}
}
```

```
export default ExampleComp;
```

- **Handling events in class components: Binding in the render() method (using arrow function as a callback)**: The use the arrow functions as a callback was introduced in ES6. When using this alternative, our event handler is automatically bound to the component instance, and we do not need to bind it in the constructor.

- **Example**:

```
import React, { Component } from "react";
class ExampleComp extends Component {
sayHi(greetName) {
console.log("Hi" + " " + greetName);//prints "Hi Abebe"
}

render() {
return <button onClick={()=> this.sayHi("Abebe")}> Click me </button>;
}
}
export default ExampleComp;
```

- **Handling events in class components: Binding at the method function (using arrow function)**: With ES7 class properties (currently supported with Babel), we can do bindings at the method definition. This approach is probably the best way of doing bindings. It's simple, easy to read, and most importantly, it works.

- **Example**:

```
import React, { Component } from "react";
class ExampleComp extends Component {
```

```
sayHi = () => {
console.log(this); /ExampleComp.js as value of  "this"
};
render() {
return     <button     onClick={this.sayHi}>     ClickMe
</button>;
}
}
export default ExampleComp;
```

- **Handling events in functional components**:

```
import React from "react";
 export default function ExampleComp() {
const sayHi = function () {
alert("hi");
};
return <button onClick={sayHi}> Click me </button>;
}
```

11.4 Updating state values in your class component: setState() method

- **Why do we need to update states in a component?** State can be updated in response to event handlers, server responses, or prop changes. While a React component can have initial state, the real power is in updating its state. After all, if we did not need to update the state, the component should not have any state.

- **How do we update our state?** The setState() method is the primary method we use to update the state of a class component. **Note**: Always use the setState() method to change the state object, since it will ensure that the component knows it has been updated and calls the render() method.

- **The setState() method**: This method is defined inside the React.Component. We inherit this method from the Component class when we call the super() in our constructor. Therefore, the method is available to all React components that use state. The setstate() function will perform a shallow merge between the new state that you provide and the previous state and will trigger a re-render of your component and all decedents.

- **How does the setState() method update state?** The main job of the setState() method is to enqueue state changes. Once it enqueues the changes, it then tells React that this component and its children, if any, need to be re-rendered with the final updated state. Meaning, the setState() method is a request to React to batch multiple setState() calls into a single update. Therefore, setState() is not an immediate command to update states. React may delay executing the etState() command based on multiple factors. One factor can be other requests it received prior to the current request. Therefore, updating state using the setState() method is asynchronous.

    - **Note**: When you change a state using the setState() method, React updates all of the child components that receive the changed state.

- **What happens after setState() method updates our state**? Once our state is updated, React simply merges the value of the updated state object into the initial state object through the process called, reconciliation.

    - **Reconciliation**: Reconciliation refers to how React updates the DOM after a change in the component. When we call setState(), React creates a new virtual DOM tree containing the reactive elements in the component and the new state. Then React figures out on how to change the component by comparing the previous virtual DOM tree with the new virtual DOM tree using a diffing algorithm (way allowing React to update only the DOM portion that needs to be changed). After comparison process is done, React goes ahead and creates a new virtual DOM that has the new

changes. Then it updates the Browser DOM with the least number of changes possible without rendering the entire DOM again

- **Syntax for setState() method**: The method takes two parameters, the updater function and the optional call back function like this: **setState(update, [callback])**
    - **The first parameter of setState(), the Updater parameter**: Updater parameter can either be an object (containing the current up-to-date state of the component) or a function (function that returns the updated value the state). **Example of setState() first parameter with an object**: It is best to use an object as the updater parameter of setState() if you have a static state value that does not depend on the previous state value. For instance, if your initial state was “Hi” and you want to change it to “Hello” upon clicking a button, the state, you can pass an object containing the new value directly to this.setState() like this: **this.setState({ valueOne: "Hello" });**

```
class ExampleComp extends Component {
    constructor() {
    super();
    this.state = {
    valueOne: "Hi",
    };
    }
    updateValue = () => {
    this.setState({ valueOne: "Hello" });
    };
    render() {
    return (
```

```
      <div>
      <button onClick {this.updateValue}>
      Click me
      </button>
      <p>{this.state.valueOne}</p>
      </div>
      );
      }
  }
```

- **Example of setState() second parameter with a function**: Simply stated, the updater function is a function that is passed to the setState() method with the updated value of the state and props Due to the asynchronous nature of setState, it is not advisable to use this.state to get the previous state within setState. When you want to apply subsequent/multiple updates to your state, you cannot rely on the value of "this.state" immediately after calling the setState(). Meaning, when your current state depends on your previous state (if for example, your state value increments by 2 whenever your button is clicked), then you cannot directly pass an object as the updater parameter for your setState(). In such cases, make your updater parameter to be a function like this: **this.setState(function(prevState, props){stateChange})**

```
  class ExampleComp extends Component {
      constructor() {
      super();
      this.state = {
      valueOne: 40,
      };
```

```
    }
    updateValue = () => {
    this.setState(function () {
    return { valueOne: this.state.valueOne + 2 };
    });
    };
    render() {
    return (
    <div>
    <button onClick {this.updateValue}>Click
me</button>
    <p>{this.state.valueOne}</p>
    </div>
    );
    }
    }
```

- **The second parameter of setState(), the optional callback**: If you want your program to update the value of a state using setState() and then perform certain actions on the updated value of state, then you must specify those actions in a function which should be the second argument of the setState(). If we do not specify such actions in a callback function, the actions will be performed based on the previous value of the state, not the latest and updated value of the state. This is because setState() has asynchronous nature whereby changes to states are not guaranteed automatically. Let us look at the 2 examples below to explain when to use setState()' second argument, the optional callback function. The examples will also explain why setState() is called asynchronous by nature.

- **Why setState() function is asynchronous and using a callback as 2nd argument to guarantee we have a state indeed updated before using it**: State updates change the virtual DOM and cause re-rendering which may be an expensive operation. Making state updates synchronous could make the browser unresponsive due to huge number of updates. To avoid these issues, a careful choice was made to make state updates asynchronous, as well as to batch those updates. In general, making setState() method was made asynchronous to make sure that the state is updated before we use the updated value in our component. Therefore, setState() will not update your state with the new state values right after you updated it. If you need to execute some function using the updated state or to generally verify if the state did indeed update correctly, make sure to pass a function as the second argument of setState() call Look at the example below. Below, you will see two example code. Assume you have a form with an input and you assign your initial value to be the value of your input. When your code is run, it will show "Abebe", the initial state. However, you want users to update the "Abebe" input value to their respective names. Whenever there is a change of name in the input, you want that changed name value to be printed in your console. The correct way to handle this scenario is to update your state first then use a callback as second argument to print this updated value. In the first example you will see the problem with not using a second argument as callback and why we cannot see the updated value in the console. In the second example, you will see how using a callback guarantees the printing of the latest value of the updated state.

- **Example 1 (Without passing the second argument to setState)**: On changing the value of the input field from "Abebe" to "Hana" or any name, the console window first prints the previous value

(Abebe) than the current value of the input field. But the current value is not equal to the value we have typed in the input field (Hana). This is because we have not declared console.log("CurrentName " + this.state.stateOne); function inside the setState due to which console.log function is getting called on previous value of input field showing the asynchronous nature of setState().

//Without passing the second argument to setState

```
constructor() {
    super();
    this.state = {
    stateOne: "Abebe",
    };
    }
    updateValue = (newValue) => {
    console.log("IntialName " + this.state.stateOne);
    this.setState({ stateOne: newValue });
    console.log("CurrentName" + this.state.stateOne);
    };
    render() {
    return (
    <div>
    <input    value={this.state.stateOne}    onChange={(e)=>
this.updateValue(e.target.value)}/>
    </div>
    );
    }
```

- **Example 1 (Without passing the second argument to setState)**: On changing the value of the input field from “Abebe” to “Hana” the console window first prints the previous value then the current value of the input field. But the current value is not equal to the value we have typed in the input field(i.e. GeeksForGeeks) this happens because we have not declared console.log(“Current name: “+this.state.name) function inside the setState due to which console.log function is getting called on the previous value of input field. It shows the asynchronous nature of setState.

// Passing the optional second parameter to setState

```
constructor() {
    super();
    this.state = {
    stateOne: "Abebe",
    };
    }
    updateValue = (newValue) => {
    console.log("BeforeStateUpdate " + this.state.stateOne);
    this.setState({ stateOne: newValue }, () => {
    console.log("AfterStateUpdate " + this.state.stateOne);
    });
    };
    render() {
    return (
    <div>
    <input
    value={this.state.stateOne}          onChange={(e)          =>
this.updateValue(e.target.value)} />
    </div>
```

```
    );

    }
```

11.5 React component's lifecycle methods (class based)

- **What is a React component's lifecycle?** React component's lifecycle is as the series of events that happen from the birth of a React component to its death. In short, React component lifecycle is the "lifetime" of a component.
- **What are the four stages a React component goes through in its lifecycle?**
    - **Initialization**: This is the stage where the component is constructed with the given Props and initial state. This is done in the constructor of a class component.
    - **Mounting**: This is the stage where the component is mounted on the DOM and rendered for the first time on the webpage. Mounting is the stage of rendering the JSX returned by the render method itself. This is basically the birth of a component.
    - **Updating**: Updating is the stage when the state of a component is updated, and the application is repainted. This is basically the growth of a component.
    - **Unmounting**: As the name suggests unmounting is the final step of the component lifecycle where the component is removed from the page. This is basically the death of a component.
- **What are the React Life Cycle Methods?** These are React's predefined functions (methods) that are invoked/invoked in different stages/ lifecycles of the component's existence. **Note:** One important thing to notice is that lifecycle methods can only be used in class components. In functional components we have other options (Hooks). But more on that in the next class. Below let us discuss **the most common lifecycle methods** we can use the different stages of a React component; ComponentDidMount(), Render(), ComponentDidUpdate() and ComponentWillUnmount()
- **ComponentDidMount()**: As the name clearly suggests, this function is invoked as soon as the component is mounted on the DOM or inserted into the index.html. Meaning, this function gets invoked once right before the render() function is executed for the first time. Common tasks that are done in this method are connecting to web APIs or JavaScript frameworks, setting Timers using setTimeout or setInterval and adding event listeners. In

the below example, when the component is mounting it is rendered with the stateOne "Abebe". When the component has been mounted, a timer changes the state, and the state becomes "Hana".

```
class ExampleComp extends Component {
    constructor() {
    super();
    this.state = {
    stateOne: "Abebe",
    };
    }
    componentDidMount() {
    setTimeout(() => {
    this.setState({ stateOne: "Hana" });
    }, 1000);
    }
    render() {
    return (
    <div>
    <h1>{this.state.stateOne}</h1>
    </div>
    );
    }
    }
```

- **Render()**: We are already familiar with the render method in React, each class component needs to contain a render method, and it is simple to understand. 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.

- **ComponentDidUpdate()**: This method is not called for the initial render, but it is every time the state of the component changes/gets updated. **Note**: componentDidMount will execute only after our index.html is rendered/generated on browser. Meaning, if we have componentDidMount() {alert("hi")}, the "hi" will display only after index.html is generated and shown on browser. **Note**: Another thing to note here is that, if we have componentDidUpdate() method called to update an initial state (assume a button is clicked and a state value has changed to "hi"), even if the component renders, the "hi" will not run every time we use componentDidUpdate().

    - **Parameters of componentDidUpdate()**: The method takes previous props, previous state and snapshot as its three parameters. Because you have previous state and props values, it allows you to compare the current value with the previous. Inside the method we can check if a condition is met and perform an action based on it.

    - **Avoid setState() componentDidUpdate()**: Never use setState() inside componentDidUpdate() because it would lead to infinite loop and cause extra re-renderong that is not visible to the user. However, if you want to call setState() in componentDidMount(), you must wrap it in a condition to avoid an infinite loop. To avoid an infinite loop, all the network requests are needed to be inside a conditional statement like below:

```
componentDidUpdate(prevProps, prevState) {
    if (prevState.data !== this.state.data) {
        // Now fetch the new data here.
    }
}
```

    - Look at the counter example we did in class to explain how to use componentDidUpdate(). In the below example, we have tow states, the "count"

and the “age” states. We have two buttons for each and when a count button is clicked, we want the count state value to increase by 1 and when the age button is clicked, we want the initial age state to increase by 10. Whenever value of the count state is changed, we also want the title of the browser document to change to that same value. Note that, we do not want the title of the document to change when age state is changed. We said we call componentDidUpdate() only after there is a change in state of a component. Here too, we will use componentDidUpdate() to change the document title only after we make sure the count state has changed. Please notice how a conditional statement is used under the componentDidUpdate() method to execute the document title update only if the count value changes (to avoid title update when age updates only)

```
class UsingClassLifeCycleComponent extends Component {
    constructor(props) {
    super();
    this.state = {
    count: 0,
    age: 5,
    };
    }
    updateAge = () => {
    this.setState(() => {
    return { age: this.state.age + 10 };
    });
    };
    updateCount = () => {
    this.setState(() => {
    return { count: this.state.count + 1 };
    });
```

```
    };
    componentDidUpdate(prevProps, prevState) {
    if (prevState.count !== this.state.count) {
    document.title = `Count: ( ${this.state.count} )`;
    }
    }
    render() {
    return (
    <div>
    <button onClick={this.updateCount}>
    COUNT:{this.state.count}</button>
    <buttonon Click={this.updateAge}>
    AGE: {this.state.age}</button>
    </div>
    );
}
}
```

- **ComponentWillUnmount()**: This is the last lifecycle method, which will be called when the component is about to be removed from the DOM tree and destroyed. When a component is removed from DOM, componentWillUnmout() will be executed before it is unmounted. **Note**: We cannot call setState() during this lifecycle method because the component is destroyed and cannot be re-rendered. Once a component instance is unmounted, it will never be mounted again. A few examples are, when you get alert on your browser saying, “are you sure you want to leave this page”, they are telling you that the component you are on is about to be destroyed, unsubscribing from a website and other cleanup routines. **Example for ComponentWillUnmount()**: Assume you have a parent

component called ExampleComp.js. In your parent component, let us say you want to show a header component, a child component, called “MyHeader.js”. You decided to show the MyHeader component in your parent component when the component is mounted originally. However, there is a delete button that you created to delete the MyHeader component from the parent component when the button is clicked.

// Parent component (ExampleComp.js)

```
constructor() {
    super();
    this.state = {
    show: true,
    };
    }
    deleteHeader = () => {
    this.setState({ show: false });
    };
    render() {
    let myHeader;
    if (this.state.show) {
    myHeader = <FufiChild />;
    }
    return (
    <div>
    {myHeader}
    <button onClick={this.deleteHeader}>
    DeleteHeader </button>
    </div>
    );
```

```
        }
```

// Child component (MyHeader.js)

```
class MyHeader extends Component {

        componentWillUnmount() {

        alert("The FufiChild component is about to be unmounted.");

        }

        render() {

        return <h1>HelloReact!</h1>;

    }  };
```