12. React Hooks

12.1 What are React hooks, why do we use them and what rules should we follow when using them?

- **Difference between class and functional component (revision)**: A key difference between class and functional components is that class components have a state while functional components are stateless. Stateless components are those components which do not have any state at all, which means you cannot use the this.setState() method inside these components.
- **Why cannot we use setState() and lifecycle methods in functional components?** A React functional component is like a normal JavaScript function with no render method. It has no lifecycle, so it is not possible to use lifecycle methods such as componentDidMount(). However, there is a Hook in React to add state behavior in functional component. Remember, any data that changes in the application is called state and when any of the data /state changes, React re-renders the component. Hooks were introduced to handle reactive data in functional components by adding state behavior and lifecycle methods to functional components.
- **What are Hooks?** Hooks are JavaScript functions that let you hook onto React state and lifecycle features inside function components. They let you use state and other React features without writing a class.
  - **Note**: Hooks are backward-co[illegible]tible, which means it does not contain any breaking changes. Also, it does not replace your knowledge of React concepts.
  - **Note**: Hooks do not work inside classes. Hooks can only live in functional components and are meant to replace the need for classes
  - **Note**: Hooks do not introduce a new concept. They just provide a more direct API to the things that you already know
- **Mismatching Versions of React and React DOM**: Hooks are the new feature introduced in the React 16.8 version, therefore, to use React Hooks, you need to have react 16.8 or above. Please run this command: npm ls react-dom

- **When do we use a Hook?** If we write a function component and realize we need to add some state to it, previously we had to convert it to a class component. Now you can use a Hook inside the existing function component to add state to your functional component.
- **Why use Hooks (the origin of React hooks)**: Let us discuss the 3 major problems React Hooks was created to solve to explain the origin of React hooks.
    - **The problem of wrapper hell in class components**: Until hooks were introduced, the main way to inject reusable logic to our component was by wrapping it in another component, usually achieved by creating a Higher Order Component (HOC) or utilizing the render props pattern. This makes the components too large.
        - With hooks, we can inject a reusable logic into our component without having to create HOC or use the render props pattern to do it. In short, with hooks no need to use classes or wrappers, component can simply use the hook that contains the logic that it needs.
    - **The confusing nature of classes**: Classes confuse both people and machines. At first, the keyword "this" can be confusing to understand. There is also the concept of bind.
        - Hooks let you use more of React's features without classes. React hooks embraces functional components and gives them access to important React features without the worry of writing complex class components.
    - **The issues with huge components**: In addition to the confusing nature of classes, when we deal with state in class components, we can end up with a huge class component with so much logic and lifecycle methods. In class components, we need to describe lifecycle methods, state of the component, component's functions/methods (that change our state). Because React does no let you sperate state managements, we can have fetches happen in componentDidMount() and componentDidUpdate(). But in that same componentDidMount(), we can see unrelated logic that deals with event listeners. In doing so, classes grow and often store different logic in one place. Since we use the life cycle methods to organize states, it is not possible to break components into smaller functionality-based pieces. Having stateful logic and unrelated code all in one place can be hard to test for bugs and errors.

    - Hooks let you split one component into smaller functions based on what pieces are related (such as setting up a subscription or fetching data), rather than forcing a split based on lifecycle methods. This avoids errors. **Example**: In class components, data fetching usually happens in the componentDidMount or componentDidUpdate life cycle methods. Event listening usually happens inside componentDidMount and event un-subscription usually happens in componentWillUnmount. Since we use the life cycle methods to organize states, it is not possible to break components into smaller functionality-based pieces. Hooks lets you do that

- **React hooks vs React classes**: In general, code that you can do in your head is code that is great. Classes in JavaScript encourage multiple levels of inheritance that quickly increase overall complexity and potential errors. Anything that allows for composition with plain functions is a win. Choosing to use stateless functional components removes the risk that comes with JavaScript classes, constructor initialization and binding functions to components.

12.2 Basic React hooks: steps to implement useState() hook

- There are around 10 hooks in React, but we will cover three of the basic hooks in this course: the useState(), useEffect() and useContext().
- **useState()**: useState() is a hook that we call it inside a functional component to add some local state to it. UseState() allows our functional components which used to be stateless become stateful by allowing us to add local state to a functional component. In short, this method lets us to Hook into React's state. **Note**: useState() hook is the most important and often used hook.
- **Steps we followed when we added the counter state to our class component (revision):** Review the CounterUsingClassState component we did in class.
  - **Step 1**: Creating class component
  - **Step 2**: Initializing the counter state to 0 inside of the constructor
  - **Step 3**: Create a function/method to change the counter state value when run. This is where we used setState() to change the counter state
  - **Step 4**: Finally attaching the function with an event in the render() method. In our case, attaching the function we created to execute when a button is clicked
- **Steps to follow to implement Hooks in a functional component:**
  - **Step 1/Create a functional component**:
  - **Step 2/ Import the method useState() from React package**: We need a way to access the React state from a functional component. It is the useState() method lets you to "Hook into" the React state. That is why we need to import the useState() method from React into your component. **Example**:
    - import React, { useState } from "react";
  - **Step 3/ Initialize state value by calling useState() at the top component:** Call the useState() Hook directly inside our functional component. When you call useState() on the first line the component becomes a stateful functional component.

```
function Example () {
        ... = useState(...);
}
```

- **Step 4/ Initialize state for the component (use destructuring)**: declare a “state variable”. Example: Let us declare a state called counter using useState(). useState().
- **Step 4/ Pass a parameter to useState() to set your sate’s initial value**: The only parameter to useState() is the initial state. No use of “this” or “this.state”. Unlike with classes, the state does not have to be an object. We can keep a number or a string as we need. **Example**: Let us pass 0 as initial state value since we want to counter to show how many times a user clicked a button:
  - const [count, setCount] = useState(0)
- **Step 5/ Update your state using the updater function**: When the hook useState(initialState) is invoked, it returns an array. The first item of this array is the current state value, and the second item of this array contains a function that updates the current state. To update the component's state, invoke the updater function setState(newState) with the new state value. The component re-renders and state receives the new value newState. **Note**: You will call the state updater function from an event handler or somewhere else. **Example**: Let us update our count state by increasing the count value by 1 whenever a button is clicked.

- **Now, recap all the above steps and display the current state**:

```
function CounterIncrementDecrement() {
        const [count, setCount] = useState(0);
        const countUpdater = () => {
                setCount(count + 1);
        };

        return (
                <div>
                        <button onClick= {countUpdater}>
                                ClickToIncreament
                        </button>
                        <h3>Count displayer: {count}</h3>
                </div>
        );
}
```

- **Note!!!! You can call/use the useStae() or useEffect() hooks multiple times in a single component**: So how does the code know which state is corresponding to which Hook? Hooks look to their order within the component to determine what state they correspond with, always. **Example**: Let us declare 2 states (age and fruit) using useState() and update both states. Because Hooks look to their oder in the component, the second we call setFruit(), it is known that only the state "fruit" will update. The other state, "age", will not update.

```
function CounterIncrementDecrement() {

        // Declaring multiple state variables

        const [age, setAge] = useState(42);

        const [fruit, setFruit] = useState("Banana");

        const changeMyFruit = (fruit) => {

                setFruit("Apple");

        };

        return (

                <div>

                        <button onClick={changeMyFruit}>

                                ClickForNewFruit

                        </button>

                        <h1>{fruit}</h1>

                </div>

        );

}
```

- **Rules to follow when using React Hooks**:
  - **Call Hooks only at the level in the body of your functional component**: Calling hooks at the top level of your functional component will guarantee that hooks are called in the same order each time a component renders. Therefore, do not call hooks inside loops, conditions, any nested function or event handlers.
  - **Call hooks only from React functional components**: You cannot use Hooks in every function, but only in React function components. You cannot even use them in regular functions within our functional components. Meaning, do not call them from any regular JavaScript function but only from React functional component.

12.3 Basic React hooks: using useState() to change a state's value based on previous state value

- **Setting state based on the previous state**: If you come across a scenario where the new state is calculated using the previous state, you can update the state with a callback, it is always a good idea to pass the value in a callback function as opposed to directly passing the state variable. We have discussed in the previous class that state update in React was deliberately made asynchronous to make sure that the state is updated before we use the updated value in our component. Therefore, your update function will not update your state with a new one right after you pass your new state in your useState(). 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 update your state with a callback. **Example**: If you look at the example below, we are going to initialize the age state with 5 as value. Let us use a callback in the updater function to add 5 to the latest age value whenever a button is clicked.

```
//Updating the state with a callback

function CounterIncrementDecrement() {

        const [age, setAge] = useState(10);

        const changeMyAge = () => {

                setAge(function (age) { return age + 5;});

        };

        return (

                <div>

                        <button onClick={changeMyAge}>

                                ClickForNewAge

                        </button>

                        <h1>{age}</h1>

                </div>

        );

}
```

- **State management with useState() summary**:
    - Call useState() hook to enable state in a functional component.
    - The first argument of the useState(initialValue) is the state's initial value.
    - useState() returns an array of 2 items: the state value and a state updater function. **Example**: [state, setState] = useState(initialValue)
    - Invoking the state updater function setState(newState) with the new value updates the state. Alternatively, you can invoke the state updater with a callback setState(prev => next), which returns the new state based on previous.

12.4 Basic React hooks: steps to implement useEffect() hook

- **Why useEffect()**: useEffect() is a React hook that allows our functional components use the component lifecycle methods (such as, componentDidMount, componentDidUpdate and componentWillUnmount) which were, in the past, only available for class components. useEffect() can be used to execute actions when the component mounts, or when certain prop or state updates or to execute code when the component is about to unmount.
- **Understanding useEffect()**: We now know that components are used primarily to compute and render outputs whenever a prop or state of the component changes. However, there are times when a component makes computations that do not target the change in state/prop value. These calculations are called side-effects. It is the useEffect() method that we use if we want to calculate side-effects independent from renderings.
- **Explaining what useEffect() does in English**: Assume we have a component that has a count state with 5 as initial value and that the state value increases by 1 whenever a button is clicked. Assume you want your component to render the updated value upon every click. The primary job of your component is to calculate the state change and display the updated state in the browser. However, let us assume you want to run a side effect (changing the title of your document) independent of what your component renders. This is when you use useEffect(). Your rendering logic will render your updated count state and your logic inside useEffect() method will change the title of your document after the component has rendered.
    - **So, what does useEffect do in short?** It allows us to tell our component to execute some side-effect logic after rendering the component.
    - **Some examples of side effects**: Fetching data, updating the DOM, and timers.
- **UseEffect() accepts two arguments**: a call back and an array containing state variables
    - **Syntax**: useEffect(callback, dependencies)
    - **The first argument is a callback function**: This is where you write your side-effect logic. Please note that the callback function (your side-effect logic) will

execute after the changes are pushed to DOM (after your component renders). **Example**:

```
function UsingUseEffect() {

const [count, setCount] = useState(0);

        useEffect(function(){

        alert ("hi")

        })

}
```

- **The second argument is an optional array of dependencies/state variables**: The dependencies argument of useEffect() lets you control when the side-effect runs. Meaning, if we use this argument, the logic in your callback (first argument) executes only if the dependencies have changed between renderings.
    - **Note**: If you do not provide the dependency argument, then, the side effect runs after every rendering.

```
function UsingUseEffect() {

        const [count, setCount] = useState(0);

        useEffect(function(){

                alert ("hi")


        })

}
```

- **Note**: If you provide an empty array ([ ]) as a dependency argument, the side-effect runs only one time after the initial rendering similar to componentDidMount() lifecycle method. This is because the execution of the effect does not depend on any state change.

```
function UsingUseEffect() {

        const [count, setCount] = useState(0);

        useEffect(function(){

                alert ("hi")



        }, [ ])

}
```

- **Note**: If you provide state or props value as the value of dependency argument, the side-effect logic runs only when the value of these specified state/props changes and the component renders. Below, the alert message will show every time there is a click that changes in the count state value

```
function UsingUseEffect() {

        const [count, setCount] = useState(0);

        useEffect(function(){

                alert ("hi")

        }, [count ])

}
```

- **Now, recap all the above explanations and see how we can use useEffect()**: Below is an example where we have a count state that will increase whenever a button is clicked. In addition to that we want to create a side effect whereby the title of the document will display the current count state value upon click.

```
function UsingUseEffect() {
        const [count, setCount] = useState(0);
        const countUpdaterFunc = () => {
                setCount(function (count) {
                        return count + 5;
                });
        };
        useEffect(function () {document.title = "Count :" + count;}, [count]
                return (
                        <div>
                                <h3>Current count is: {count}</h3>
                                <button onClick={countUpdaterFunc}>
                                        ClickToIncreaseCount
                                </button>
                </div>
            );
        }
```

- **Avoiding delay in performance by configuring useEffect() hooks to run when a specific state/props change happens only**: If we have multiple state variables that exist inside a component, change to any of these state variables leads to the re-rendering of the component. By default, useEffect() runs during initial rendering as well as during the re-rendering of the component. That means, on each re-rendering, all of our useEffect() hooks are invoked by default, though we do not need to run useEffect() for each state change/or for each component rendering. This slows down the performance of your application. To avoid this, we can configure our useEffect() to execute only when specific states/props update. **Example**: Below, let us create a component that has count and age states where the count state increases by 1 when a button is clicked and the age state increases by 1 when another button is clicked. In addition, we want to show 3 side effects after the component renders. Right after the component renderes, we want to show an alert side effect that displays an alert message of "Component is mounted". We also want to show another effect that changes the background color of the body to green only if the count state changes. We also want to show a 3rd effect whereby we change the title of the document to reflect the current age value only if the age state changes. ***In short, we want the alert effect to run for every component render, we want the background color effect to run only when count changes and we want the title effect to run only when the age state changes***.

```
function UsingUseEffect() {
        const [count, setCount] = useState(0);
        const color = count % 2 === 0 ? "white " : "green";
        const [age, setAge] = useState(0);
        const changeCounter = () => { setCount(() => count + 1);};
        const changeAge = () => {setAge(() => age + 5);};
        useEffect(() => {alert("Component is mounted");}, [ ]);
        useEffect(() => {
                document.body.style.backgroundColor = color;
        }, [color]);
        useEffect(() => { document.title = age;}, [age]);
        return (
                <div>
                        <h1>Count : {count}</h1>
                        <buttononClick={changeCounter}>
                                ClickHere
                        </button>
```

```
                    <h1>Age : {age}</h1>
                    <button onClick={changeAge}>ClickHere</button>
                </div>
            );
        }
```

- **useEffect cleanup function**: Some effects require cleanup to reduce memory leaks. Timeouts, subscriptions, event listeners, and other effects that are no longer needed should be disposed. We do this by including a return function (commonly known as useEffect cleanup function) at the end of the useEffect() hook. The useEffect cleanup is a function in the useEffect hook that allows us to tidy up our code before our component unmounts. When our code runs and reruns for every render, useEffect also cleans up after itself using the cleanup function. Note: **Note** Do not update the state inside the return/cleanup function.
- **Syntax for useEffect cleanup function**:

```
useEffect(() => {
    //Effect code here
    return () => {
      Cleanup code
    }
  }, [stateDependency])
```

- **Scenarios where and when the useEffect cleanup function useful**: Assume we get a fetch of a particular user through a user’s id, and, before the fetch completes, we change our mind and try to get another user. At this point, the id, updates while the previous fetch request is still in progress. This is because useEffect() always attempts to update state of a component even if the component has been unmounted. As a result, we will get a warning message that states “***Warning: Can’t perform a React state update on unmouted***

component”. Therefore, it is then necessary for us to abort the fetch using the cleanup function, so we do not expose our application to a memory leak.

- **Example for useEffect() cleanup function**: Let us assume that you have a parent component called UsingUseEffect.js (that has count as its state) and you want to display 2 child components (Hello.js and Bye.js) in it conditionally. Meaning, assume you want to display <Hello/> child component in your UsingUseEffect.js if your count state is updated to a value called “even” when a button is clicked. Assume again that you want to display the <Bye/> child component if the count state is updated to “odd” upon a click on another button. The <Bye/> component only renders the “bye” message. The <Hello/> component has a state called greet and uses useEffect() to display the effect “hello” message after counting 3 seconds whenever the greet state value updates to “hello”.
  - Assume that while the <Hello/> component was counting 3 seconds and before it finishes counting to display the “hello” message, you clicked the odd button. This is when above “*Can’t perform a React state update on an unmounted component*” warning message comes in your console. This warning is because the <Hello/> component has already been unmouted, but the useEffect() in this component still is trying to update the greet state of the unmouted component. That is why we need to cancel any side-effect if a component unmounts.
  - To cancel the side-effect, we said we can return a cleanup function within our useEffect() in our <Hello/> component. Basically, we need to clear the timer using clearTimeout() built in function. Look below for explanation with example:

}

//Parent component

```
function UsingUseEffect() {
        const [count, setDisplay] = useState("even");
                return (
                        <div>
                                <button onClick={() => { setDisplay("even");}}>
                                        Even
                                </button>
                                <button onClick={() => { setDisplay("odd");}}>
                                        Odd
                                </button>
                                <p>{count === "even" ? <Hello /> : <Bye />}</p>
                        </div>
                );
}
```

// Child component (Bye.js)

```
 function Bye() {
        return <div>Bye</div>;
}
```

// Child component (Hello.js)

```
function Hello() {
const [greet, setGreet] = useState(" ");
        useEffect(() => {
                let timer = setTimeout(() => {
                        setGreet(() => "Hello");
                }, 3000);
```

//   cleanup function

```
                return () => clearTimeout(timer);
        }, []);
        return (
                <div>
                        <h1>{greet} from Hello.js</h1>
                </div>
        );
```

```
}
```

- **useEffect() explained in terms of lifecycle methods (summary)**: When we started discussing about useEffect() above, we said that React's useEffect() hook combines componentDidMount, componentDidUpdate and componentWillUnmount lifecycle methods. We also said that using this hook, as opposed to the lifecycle methods, reduces and simplifies our code, and also allows for multiple useEffect() hooks to be called in a single component. Now, let us discuss how we can achieve all lifecycle methods (provided by class components) using useState() hook in our functional component.
    - **componentDidMount() equivalent**: We said that the componentDidMount() function gets invoked as soon as the component is mounted on the DOM or inserted into the index.html. To achieve this using useEffect() hook, all we have to do is make the hook run only once (i.e., when the component is mounted on the DOM). To do this, we just need to set an empty array as a hook dependency. This is because on the initial mount of the component, useEffect() runs the callback function without checking if the dependency has changed or not.

      ```
      useEffect(() => {
                /* Write your ComponentDidMount code here */
      }, [ ]);
      ```

    - **componentDidUpdate()equivalent**: We said componentDidUpdate() method gets called every time when the state of the component updates. To have useState()

hook run when the component is updated, all that we need to do is set at least one dependency state variable for the hook to run. This is because when the state changes and the component updates, the dependencies array is checked (before running the useEffect() function) to see if it changed since the last update

```
useEffect(()=>{
        /* Write your ComponentDidUpdatecode here */
}, [state1]);
```

- **componentWillUnmount() equivalent**: We said previously that this method will be called when the component is about to be removed from the DOM tree and destroyed. We also said that this method is a good place toto do some cleanup works like invalidating timers or cleaning up any subscriptions that were created in componentDidMount(). To have useEffect() hook run when the component is unmounted, all we need to do is return a function from the hook. Returning an anonymous function with useEffect() will it run before the component unmounting. The empty array passed as second argument below tells to useEffect() to run only once when the component is unmounted.

```
useEffect(() => {
          return () => {
                    /* Write your componentWillUnmount() code here */
        }
}, [ ]);
```

12.5 Basic React hooks: steps to implement context API and useContext() hook

- **Questions to ask before diving into React Context API**: See below for answers to these questions.
  - How do you send data to child components?
  - How do we pass data from a parent component to a grandchild?
  - How about if we directly want to pass data to the grandchild component without going through the child?
- **What is the problem React Context API was created to solve?** Prop drilling is the problem React Context was created to solve
  - **Props drilling**: It is the process of sending data (props) from a higher-level component to several nested children (lower level) components. The major problem with prop drilling approach to share data is that most of the components through which this data is passed to are forced to take in unrelated data just for the sake of passing it to the next component. Thus, these intermediary components will be forced to re-render, degrading the app's overall performance. In this section, we will explore how we can use React Context API to avoid prop drilling.
  - **Props drilling explained in example**: In a typical React application, data is passed top-down (example, from parent component to child component) via props. Assume we have one parent component with several child components. Assume

the child components have children of their own. What if the children to exchange data with other children (siblings)? The only way to pass data to a component's sibling was to move state to a higher (parent) component and then pass it back down to the other sibling via props. However, passing state down from top-level components can be cumbersome and time-consuming. In addition, components that do not need to use the state data will still have to access it and pass it down as props. Prop drilling for certain types of globally used data (such as paddings and font-sizes that are shared by many components in an application) is laborious and time-consuming. React Context was introduced to allows us to pass data through our component trees, giving our components the ability to communicate and share data at different levels. So, React context was created to fix these issues.

- **React Context API**: Context is a way to share global data (like props, state or variables) between deeply nested components more easily without the need to manually pass props down to each nested component. **Note**: Each component in Context is context-aware, therefore, instead of passing props down through every single component on the tree, the components in need of a prop can simply ask for it, without needing intermediary helper components that only help relay the prop.
- **How does Context API work?** The Context API basically lets you broadcast your data to multiple components by wrapping them with a context provider. It then passes this data to the context provider using its value attribute. Then child components can tap into this provider using a context consumer or the useContext() Hook when needed. **Note**: We will discuss about consuming data using useContext() hook later.
- **Steps to use context API**:
  - **Step 1**: create a context using the createContext() method from React and save it on a variable. Make sure to export the context you created because in most cases your component will be in another file. **Example**:

    ```
    const MyContext = React.createContext()
    ```

  - **Step 2**: Take your created context and wrap the context provider around the child components you want to pass down data to. **Note**: The context you created above, the “MyContext” , is an object that has two properties, namely Provider and

Consumer. Both Provider and Consumer are components. So, go ahead and wrap your Provider component around your child component. **Example**:

```
<MyContext.Provider>
    <ChildComponent/>
</MyContext.Provider>
```

- **Step 3**: Put the value you want to pass to any child components on your context provider using the value prop. **Example**:

```
<MyContext.Provider value={{ user: "Alem" }}>
    <ChildOne />
</MyContext.Provider>
```

- **Step 3**: Use the Consumer component to use/consume/read the value of the context (which you created above) in any child component. **Note**: To consume the passed down value, we use a technique called "render props". Render props is a technique for sharing data/code between React components using a prop whose value is a JavaScript function. So, use the Consumer component to wrap this function and make this function to return the value of your context. **Example**:

```
<MyContext.Consumer>
{function (value) {
                        return <div>{value}</div>;
                        }
                }
</MyContext.Consumer>
```

- **Examples on how to use Context API**: Before looking at an example on how to avoid prop drilling using the context API, it is better to see an example on how to pass data to child components using prop drilling. For both examples below, assume that we are building an app that welcomes a user by first name when the user logs in. Assume that we have the root/parent component, called Parent.js where the user's name object (props) is available in. However, the component that renders the welcome message with the user's name is nested deep within our child component called, ChildThree.js. The user's name object will need to be passed down to ChildThree.js component through two other child components called, ChildOne.js and ChildTwo.js. Now, let us see how we can pass the user's name data from Parent.js to ChildThree.js using prop drilling and context API.

- **1. Example of passing data from parent to a child in a deeply nested component (prop drilling)**:

// App.js

```
function App() {
        return (
                <div>
                        <MyParent />
                </div>
        );
}
```

// Parent.js

```
import React, { useState } from "react";
import ChildOne from "./ChildOne";
function MyParent() {
        const [user, setUser] = useState("Alem");
        return (
            <div>
                <ChildOne user={user} />
            </div>
        );
    }
    export default MyParent;
```

// ChildOne.js

```
    import React from "react";
    import ChildTwo from "./ChildTwo";
    function ChildOne(props) {
        return (
            <div>
                <ChildTwo user={props.user} />
            </div>
        );
    }
    export default ChildOne;
```

// ChildTwo.js

```
    import React from "react";
```

```
import ChildThree from "./ChildThree";

function ChildTwo(props) {

        return (

                <div>

                        <ChildThree user={props.user} />

                </div>

        );

}

export default ChildTwo;
```

// ChildThree.js

```
import React from "react";

 function ChildThree(props) {

        return <div>Welcome : {props.user}</div>;

}

 export default ChildThree;
```

- **2. Example of passing data from parent to child in a deeply nested- component (context API)**:

// App.js

```
import React from "react";

import "./App.css";

import MyParent from "./Components/UseContextEdom/MyParent";
```

```
function App() {
    return (
        <div>
            <MyParent />
        </div>
    );
}
export default App;
```

//MyParent.js (this is where you provide a context to allow data to pass down to child components)

```
    import React from "react";
    import ChildOne from "./ChildOne";
    export const MyContext = React.createContext();
    function MyParent() {
return (
    <div>
    <MyContext.Provider value={"Alem"}>
    <ChildOne />
    </MyContext.Provider>
    </div>
);

}
```

// ChildOne.js

```
import React from "react";

import ChildTwo from "./ChildTwo";

 function ChildOne() {

return (

<div><ChildTwo /></div>

);

}
```

// ChildTwo.js

```
import ChildThree from "./ChildThree";

 function ChildTwo() {

return (

<div><ChildThree /></div>

);

}
```

// ChildThree.js (this is where you consume a context to use/consume the data passed down from MyParent component)

```
import { MyContext } from "./MyParent";

 function ChildThree() {
```

```
return (
<div>
<MyContext.Consumer>
{function (value) {
return <div>{value}</div>;
}}
</MyContext.Consumer>
</div>
);
}
```

- **Major drawbacks of depending on the Context API**:
  - **Problems with component reusability**: When a context provider is wrapped over multiple child components, we are implicitly passing whatever state/data that is stored in that provider to all child components it wraps. Please note that the data/state is not literally passed the child components until we initiate an actual context consumer (or useContext() hook). However, by providing a context, we are implicitly making these child components dependent on the state provided by this context provider. We will encounter a problem when we try to reuse any of these components outside the context provider. For instance, if we do not want to wrap a child component in the Context Provider and reuse it for some other purpose, the component first tries to confirm whether that implicit state provided by the context provider still exists before rendering. When it does not find this state, it throws a render error. Remember our previous example and say in our MyParent component, oour ChildOne component was placed outside our context provider and you wanted to reuse the ChildOne component to display a different

message based on a different condition. In this case, ChildThree, the component that will use the state from parent component, will confirm if the state (provided by the context provider) still exists before it renders. It will not find the state as ChildOne component is placed outside the context provider. Therefore, it throws a render error.

  - **Problems with performance**: The Context API uses a comparison algorithm that compares the value of its current state to any update it receives, and whenever a change occurs, the Context API broadcasts this change to every component consuming its provider, which in turn results in a re-render of these components. This would seem trivial at first glance, but when we rely heavily on Context for basic state management, we needlessly push all of our states into a context provider. As you would expect, this is not very performant when many components depend on this Context Provider, as they will re-render whenever there is an update to the state regardless of whether the change concerns or affects them or not.

- **Why do we then need the useContext() hook?** We have seen above how we can use Context API to avoid prop drilling and send data from parent components to children. To make data consuming easy, React 16.8 introduced the useContext() hook. When we use the useContext() hook, we do not need the render props technique (we used under Context API). Rather, we can pass the entire context object (which we created in the parent component) into our React.useContext() hook on top of our consuming child component. In short, the useContext() hook was introduced to simplify the consuming part of context API.
- **Steps to use useContext()**: Step 1, Step 2 and Step 3 we used for Context APU above remain the same when using the useContext() hook. However, the consuming step is diffenent here.
  - **Step 1**: create a context using the createContext() method from React and save it on a variable. Make sure to export the context you created because in most cases your component will be in another file. **Example**:

```
const MyContext = React.createContext()
```

- **Step 2**: Take your created context and wrap the context provider around the child components you want to pass down data to. **Note**: The context you created above, the "MyContext" , is an object that has two properties, namely Provider and Consumer. Both Provider and Consumer are components. So, go ahead and wrap your Provider component around your child component like below. **Example**:

```
<MyContext.Provider>
        <ChildComponent/>
</MyContext.Provider>
```

- **Step 3**: Put the value you want to pass to any child components on your context provider using the value prop. **Example**:

```
<MyContext.Provider value={{ user: "Alem" }}>
        <ChildOne />
</MyContext.Provider>
```

- **Step 4**: Go to the child component you finally want to consume your data/value. Import the useContext() hook from React. Also, import the context your created in your parent component. **Example**:

```
import React, { useContext } from "react";
 import { MyContext } from "./MyParent";
```

- **Step 5:** In the above child component, on top of the component, call the useContext () hook and pass the entire context object as its argument and put this value on a variable. **Example**:

```
const myValue = useContext(MyContext);
```

- **Step 6**: In the above child component, consume/read the value of the context in your child component. To do this, make your component return the above ("myValue"). **Example**:

```
return <div>Welcome : {myValue}</div>;
```

- **Example on how to easily consume data using the useContext() hook**:

// App.js

```
import React from "react";
import MyParent from "./Components/UseContextEdom/MyParent";
function App() {
return (
<div>
<MyParent />
</div>
);
}
```

// MyParent.js

```
import React from "react";
import ChildOne from "./ChildOne";
export const MyContext = React.createContext();
function MyParent() {
return (
<div>
<MyContext.Provider value="Alem">
<ChildOne />
</MyContext.Provider>
</div>
```

```
        );
        }
    // ChildOne.js
        import React from "react";
        import ChildTwo from "./ChildTwo";
         function ChildOne() {
        return (
        <div><ChildTwo /></div>
        );
        }
```

// ChildOne.js and ***// ChildTwo.js*** headings are printed bold italic.

```
    // ChildTwo.js
import React from "react";
        import ChildThree from "./ChildThree";
        function ChildTwo() {
        return (
        <div>
        <ChildThree />
        </div>
        );
        }
         export default ChildTwo;
```

// ChildThree.js

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

import { MyContext } from "./MyParent";

 function ChildThree() {

const myValue = useContext(MyContext);

return <div>Welcome : {myValue}</div>;

}
```