What are React Hooks? (pdf)

1. What are React Hooks?

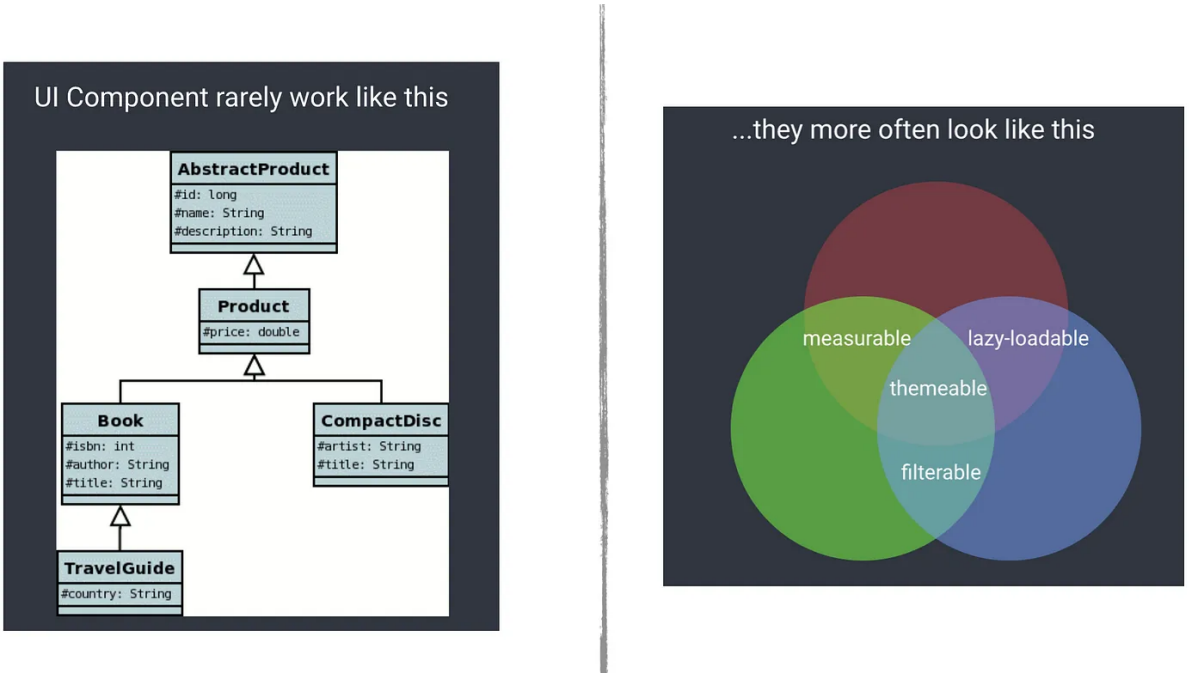
What does hook mean? Hook. Throwing a hook somewhere in the code. In other words, it means giving our component power and capability by hooking into a sharing set of functions.

If you are familiar with **Aspect Oriented Programming** or **Cross-Cut**, maybe these concepts will not be foreign to you. Some capabilities need to be used commonly by components. There are four ways to share a standard capability's code logic.

- Inheritance
- High Order Component
- Render Props
- Hooks

Inheritance logic in code sharing is not suitable for UI structures but more suitable for Model structures.

Therefore, there are three logical methods for UI in code sharing. These are HoC, Render Props, and Hooks. These methods have advantages and disadvantages over each other. I will be addressing these in the following topics.



https://www.dotconferences.com/2019/12/evan-you-state-of-components
(UI Components Works)

What we call standard code is that there are common codes such as measurable, themeable, and filterable. Well, the same algorithm and the same business logic will be operated. How about if we developers write it in Functions (Capabilities), components hook into them, and use them?

Note
Immediately, what's the difference between writing Util JS and calling a function from it? And as we will see in the future, the parts we are working on here are not just code; we also need to use the APIs that React provides us in these functions to create Views.

React also has the idea of developing its components as more light-weight function components. When necessary, they want to use to take advantage of the libraries they need; that is, they want to use the structures where you hook the relevant function to use.

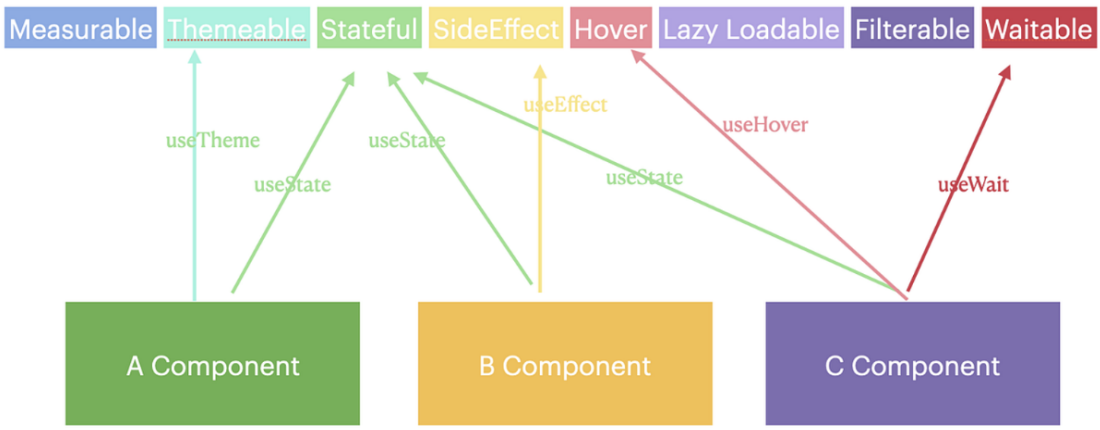
Let's do some brainstorming. React UI Components we will need what kind of capabilities.

- Keeping State (useState)
- Reflect Side Effect (useEffect)

- Hover (useHover)
- LifeCycle (, useDidUpdate)

  useDidMount
- Redux Store (useSelector, useDispatch, useStore)
- React Router (useHistory, useLocation, useParams, useRouteMatch)
- Data Fetch (useFetch)
- Theme (useTheme)
- Wait (useWait)
- Screen Dimensions (useWindowDimension)
- We can think of many common capabilities like useReducer, useRef, etc.

In my head, I have a structure similar to the picture below. Components assign Hooks to these capabilities with use hooks



Note: We could give these capabilities to components with High Order Functions or Render Props, Chain methods without using inheritance in Class Component (Components). That's right; we have been using these methods long. But this has two disadvantages.

- The code that wraps components like **h(g(f(x)))** reduces the readability of the code too much and needs to be executed as a chain sequence (pipe).
- Secondly, let x=view; for example, f, g, and h are not enough to be just functions, so we will add capabilities. Due to the React structure, they must be a component, and the **render** function must be called. However, none of the

abilities I mentioned above are related to rendering (drawing on the screen). Still, it needs to be able to render the subcomponents it contains in the tree structure.

2.1 What kind of components are there in React?

We mentioned that there are two types of React Component creation types. The first one is

- Class Component (if you want to keep State) -> class. .... extends React. Component
- Function Component (if we don't need state retention) -> function ...

How do we provide a functional method to support Function Component state retention and other lifecycles?

Then we introduced a third type, Hook, and discussed how Hook constructs lead developers to use Functional Components.

2.2 What is the problem with the Class Component?

Constructor super(props) invocation requirement

Dan Abramov touched on this issue in his blog Why Do We Write super(props)? In summary, after calling the JS constructor, you can only access it through this. You need to make this call even if you are not using any props here. But if you don't create a constructor, you can still access this.props in the renderer because React Class already binds these props when creating the Component.

```
class Checkbox extends React.Component {
  constructor(props) {
    super(props);
    this.state = { isOn: true };
  }
  // ...
}
```

constructor from https://overreacted.io/why-do-we-write-super-props/

Since the goal has always been to write components with less code, they thought about how to remove this usage. Thanks to the ES class fields proposal capability,

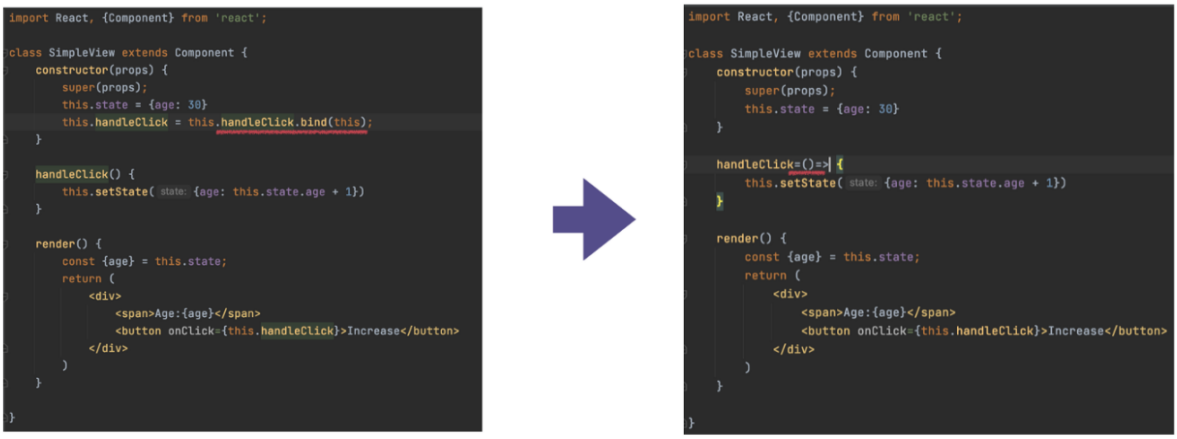
defining this in the constructor is no longer necessary, so this is not the main issue that bothers us.

```
class Checkbox extends React.Component {
  state = { isOn: true };
  // ...
}
```

constructor from https://overreacted.io/why-do-we-write-super-props/

Action Handler Bind Requirement

This issue is due to the JS language structure, and we can eliminate this binding issue by using Arrow functions. But when there was no Arrow Function capability in JS language, we had to bind the handler in the constructor.



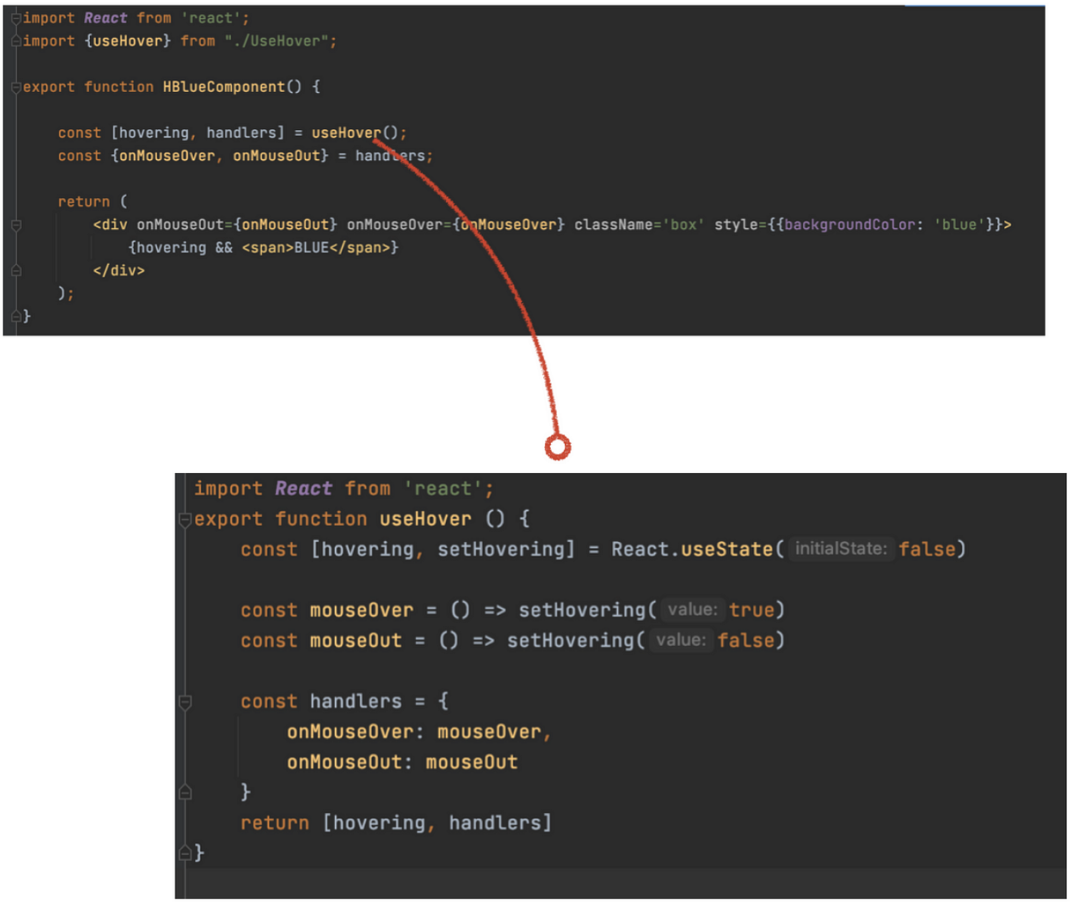
Binding Event Handling

Code Repetition Prevention Method (DRY)

We know that Inheritance, High Order Components, or RenderProps methods provide a code share over View, and when these code shares increase, the wrap methods make the code very complex.

```
withRouting
      (withStateManagement
                  (withLogging
                        (withPersisting(View)))
```

In this regard, writing Custom Hook and composing it with use provides a more readable code. It also removes this complexity.

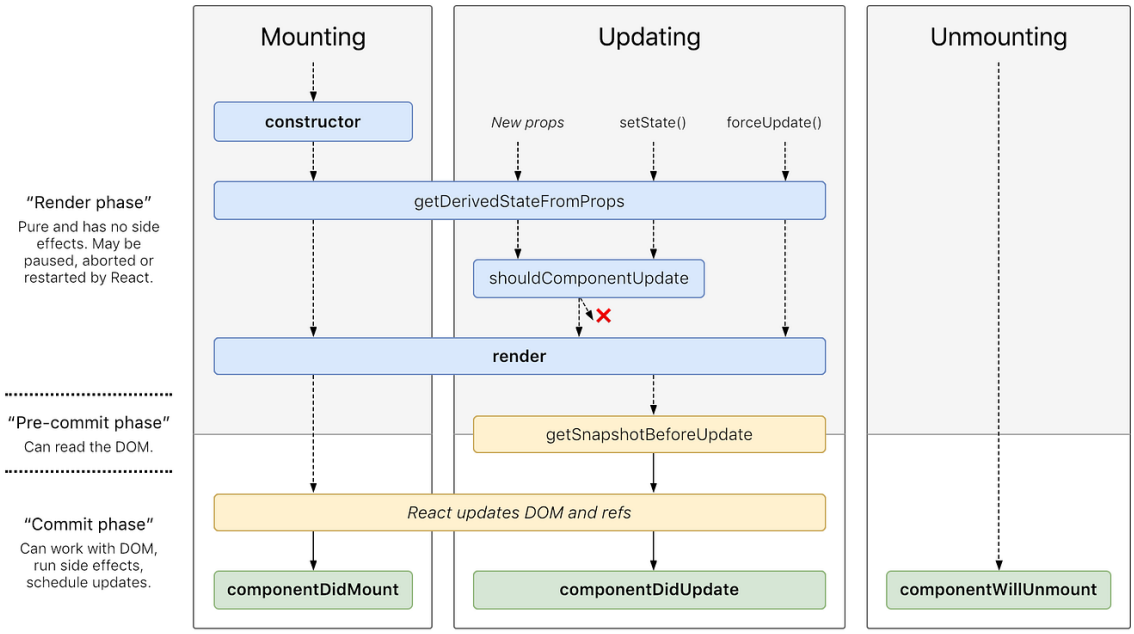


Hooks Yöntemi

The number of RenderProps helixes is actually not much. But for those who started to learn to React with Hook and Function Components, Class Components and these concepts may seem complicated.

LifeCycle in Class Components

Knowing the life cycle of the components and trying to synchronize with the existing React Rendering by understanding requires going into some technical details. It is necessary to master the LifeCycle, Virtual DOM, and Browser Rendering Pipeline topics I explained in React Ref topic.



React LifeCycle

As you can see from the image below, we need to call **componentDidMount and componentDidUpdate** separately, even if a common function is called, so that there is no function duplication; the developer is expected to understand the Component Lifecycle. This makes the React learning curve difficult.

However, React Hooks makes it more component and real-world thinking, so it does it through **useEffect**. If the developer knows that the component affects external components and its environment, knowing that it will do this in **useEffect** makes the code more understandable.

```
class ClassComponent extends Component {

    constructor(props) {
        super(props);
        this.state = {age: 0}
    }

    componentDidMount(){
        document.title = `Your age ${this.state.age} `;
    }

    componentDidUpdate(){
        document.title = `Your age ${this.state.age} `;
    }

    render() {

        const {age}=this.state;
        const {size,className}=this.props;
        return (
            <div>I am a ClassComponent {className} size {size} age {this.state.age}
                <button onClick={()=>{this.setState({age:age+1})}}>Increase Age</button>
            </div>
        )
    }
}
export default ClassComponent;
```

```
import React, { useState , useEffect}  from 'react';

const FunctionComponent = (props) => {

    const [age, setAge] = useState(0);

    useEffect(() => {
        document.title = `Your age ${age} `;
    });

    return (
        <div>I am a FunctionComponent {props.className} size {props.size} age {age}
            <button onClick={() => setAge(age + 1)}>Increase Age</button>
        </div>
    )
}

export default FunctionComponent;
```

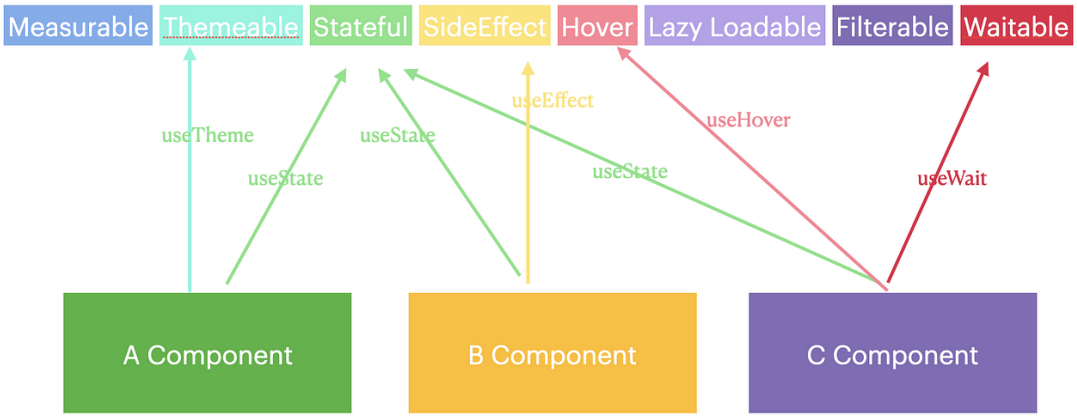
Class Component vs. Hook Component

The JS world is increasingly integrating more and more Functional Programming approaches.

Class Component class creation, using new structures, both performance and code development habits Java, .Net structural, hierarchical image is avoided because it can turn it into.

3. What is useState?

I explained that React Hook provides an shared set of behaviors for components by hooking them, just like in the picture below.



Hooking of functional components with different hooks…

One of the most critical requirements of components is to keep and use their internal state somewhere. When we write React Class Component, if we say **extends React.Component**, we derive our component from a structure with a state mechanism, but when we make a Functional Component, we can provide this with useState Hook.

The two codes in the picture below do the same job. It holds the age value in the component, which increases as this button is pressed.

Class Component

Hooks Component

Class vs. Hook Component

How do we do this with useState in the Functional Component? Let's take a closer look at the code.

```
import React, {useState} from 'react';

function HookComponent() {


    const [age, setAge] = useState(0);

    return (
        <div>
            Hook Component
            <br/>
            <span>Age:{age}</span>
            <button onClick={() => setAge(age + 1)}>Increase Age</button>
        </div>
    )

}


export default HookComponent;
```

Hooks Component

useState(0): A tool that connects the component to the infrastructure of the React library, which allows us to manage the component globally and triggers a re-render according to the state. useState comes from the react library in summary. 0 is the state default value we will keep…

[age,setAge]: We access the value held here; for example, we read (read) over age. And we write (write) over setAge

And you need to define each separate state you will use in the component in this way. In this way, we ensure that the component can be used by persisting the state during the rendering phase and that the relevant component is rendered again when the state changes.

Let's explain the subject in more detail with a few examples. In the example below, you want to keep the Theme state and accordingly render the background according to Dark or Light Mode. In this case, the state we need to keep is the **theme state.**

```
const [theme, setTheme] = React.useState('light')
```

```
function ThemeButton() {
  const [theme, setTheme] = React.useState('light')

  const toDark = () => setTheme('dark')
  const toLight = () => setTheme('light')

  return (
    <div className={theme}>
      {theme === "light"
        ? <button onClick={toDark}>🔨</button>
        : <button onClick={toLight}>💡</button>}
    </div>
  )
}
```

Or, if we look at another example, we want to make a simple ToDO app. This time our component needs to hold more than one value. In this case, these variables

- **todos** todo list of elements,
- **input**, which holds the value that the user is currently typing in Input, and
- a structure that allows us to generate an **id** before each input todos is added so that we can do the remove operation through this id.

```
const [todos, setTodos] = React.useState([])
const [input, setInput] = React.useState('')
```

```
const [id, setId] = React.useState(0);
```

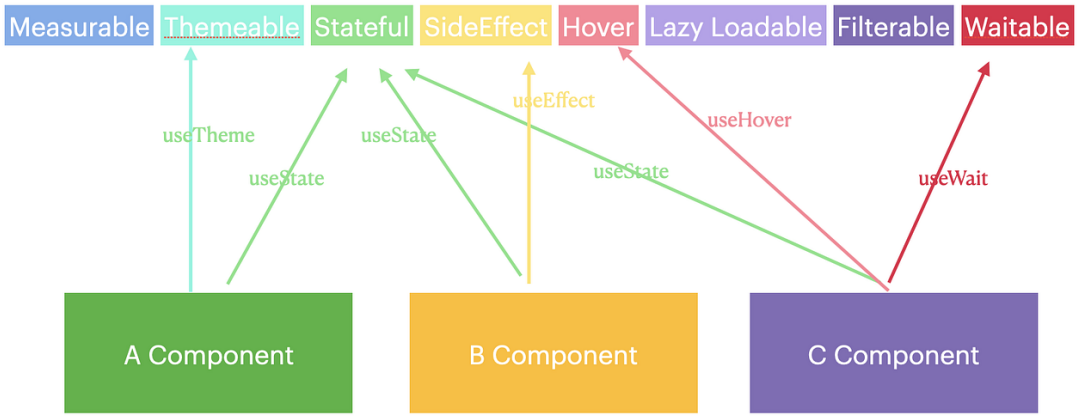
```
export function Todo() {
    const [todos, setTodos] = React.useState( initialState: [])
    const [input, setInput] = React.useState( initialState: '')
    const [id, setId] = React.useState( initialState: 0);

    const handleAdd = () => {
        setTodos( value: (todos) => [{text: input, id: id}, ...todos])
        setInput( value: '')
        setId( value: id + 1);
    }

    const removeTodo = (id) => setTodos( value: (todos) => todos.filter((t) => t.id !== id))

    return (
        <div>
            <input
                type='text'
                value={input}
                onChange={(e : ChangeEvent<HTMLInputElement> ) => setInput(e.target.value)}
                placeholder='New Todo'
            />
            <button onClick={handleAdd}>Add</button>

            <ul>
                {todos.map(({text, id}) => (
                    <li key={id}>
                        <span>{text}</span>
                        <button onClick={() => removeTodo(id)}>X</button>
                    </li>
                ))}
            </ul>
        </div>
    );
}
```

You can think of many such examples. In summary, since keeping the state is the most basic need of the components, the more examples you make about it, the more they will help you improve.

4. What is useEffect?

4.1 What is it?

I explained that React Hook is a common set of behaviors for components that are provided by hooking them, just like in the picture below.



In this article, we manage the side effect concept in functional programming in **useEffect** Hooks for React Views.

Suppose you are accessing the Network API from React Components. In that case, if you are accessing the DOM, that is, if you want to have an effect outside this component via WebAPI or with the libraries offered, you need to do this in the useEffect function in the React library.

- Adding a new effect.
- Skip making the effect take effect in the next case
- Cleaning the Effect

As for the structure, you pass a callback function when calling **useEffect** from the React library; this callback function is called (invoked) after each component rendering.

In the example below, since the console.count and document.title affects Console and Document in the Web API other than this component; we make these calls in useEffect.

```
export function Counter () {
    console.count( label: 'Counter Func Call')
    const [count, setCount] = React.useState( initialState: 0)

    React.useEffect( effect: () => {
        console.count( label: 'In useEffect, after render')
        document.title = `Count: ${count}`
    })

    return (
        <>
            <button onClick={() => setCount( value: (c) => c - 1)}>-</button>
            <h1>Count: {count}</h1>
            <button onClick={() => setCount( value: (c) => c + 1)}>+</button>
        </>
    )
}
```

By debugging the execution of this code we can understand a bit of the logic. For example, we refreshed the screen. The Counter function was called two times, the related state connection was established with useState and useEffect, and the side effect callback function was set.

When the 2nd Counter Func Call was written, the same function was called again on top of the previous callback function, and the returned JSX was printed on the screen.

After updating VDOM → DOM, the component SideEffect, i.e., useEffect, is now called.

```
Counter Func Call: 1
Counter Func Call: 2
In useEffect, after render: 1
```

Then when I increase the number by one, the component will be rendered again. But the component render is not done after the first call of the function but after the 2nd call…

The reason for this is that if we want to change the useEffect according to the callback state. Calling a function that does not render once in a while gives us this advantage.

For example, let's update our function as follows. Let's want to apply different effects on the values according to count %2

```
export function Counter() {
    console.count( label: 'Counter Func Call')
    const [count, setCount] = React.useState( initialState: 0)


    React.useEffect( effect: count % 2 === 0 ? () => {
        console.count( label: 'In useEffect, after render')
        document.title = `Count: ${count}`
    } : () => {
        console.count( label: 'In useEffect XXX, after render')
        document.title = `Count XXX: ${count}`
    })

    return (
        <>
            <button onClick={() => setCount( value: (c) => c - 1)}>-</button>
            <h1>Count: {count}</h1>
            <button onClick={() => setCount( value: (c) => c + 1)}>+</button>
        </>
    )
}
```

We updated according to 2% of Counter code.

As you can see below, the extra call of the Function allows us to create different business logic here, where we can sometimes call useEffect and sometimes useEffect XXX.

```
Counter Func Call: 1
Counter Func Call: 2
In useEffect, after render: 1
Counter Func Call: 3
Counter Func Call: 4
In useEffect XXX, after render: 1
Counter Func Call: 5
Counter Func Call: 6
In useEffect, after render: 2
Counter Func Call: 7
Counter Func Call: 8
In useEffect XXX, after render: 2
```

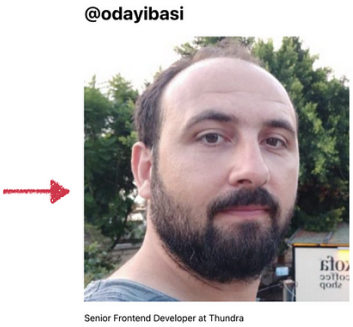
Counter kodunun %2 göre Cıktısı…

4.2 If State is Updated through useEffect

Let's examine our code below. We will pull data from the GitHub API for GitHub Profile information. There is an interesting situation here...

We are calling **setProfile** from useEffect, so we are updating the state. What effect does this have?

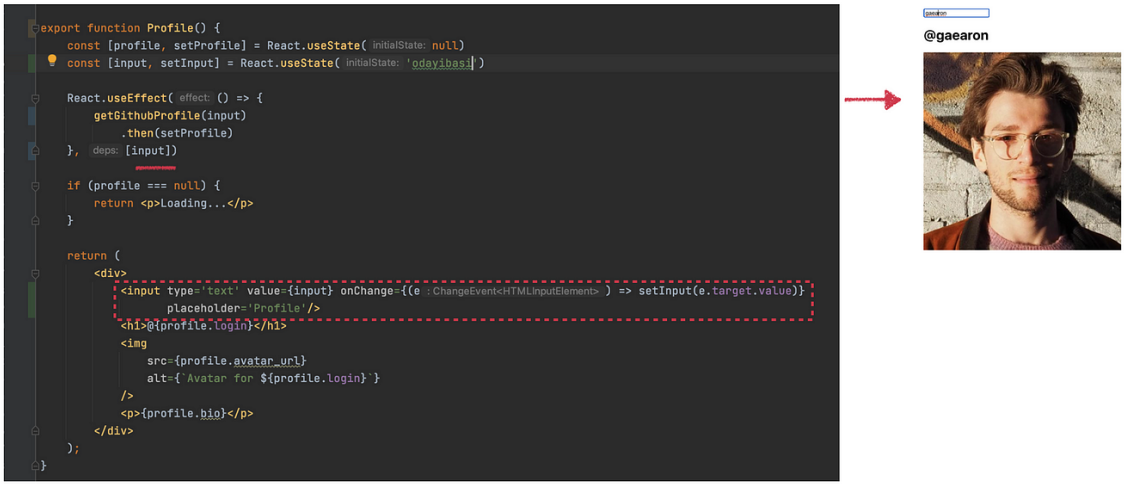
```
function getGithubProfile(username) {
    return fetch( input: `https://api.github.com/users/${username}`)
        .then((res : Response ) => res.json())
}

export function Profile() {
    const [profile, setProfile] = React.useState( initialState: null)

    React.useEffect( effect: () => {
        getGithubProfile( username: 'odayibasi')
            .then(setProfile)
    }, deps: [])

    if (profile === null) {
        return <p>Loading...</p>
    }

    return (
        <div>
            <h1>@{profile.login}</h1>
            <img
                src={profile.avatar_url}
                alt={`Avatar for ${profile.login}`}
            />
            <p>{profile.bio}</p>
        </div>
    );
}
```



In this case, setState will update the DOM and call useEffect, and useEffect will call the relevant profile from the GitHub API again. As a result, the state will be updated again, etc... Our function will be in a continuous **Cyclic Infinitive Loop**.

To prevent this, we give a scope at the end of useEffect. When we give an empty array **[],** useEffect is called once regardless of what happens after the DOM is first drawn, which we call Initial Render.

If we add Input to the component and say that useEffect should be called according to the change with the value from here, we need to give a scope here **[input].** Since we give **[input]** useEffect as scope/dependency, the call will only occur when **input** changes.

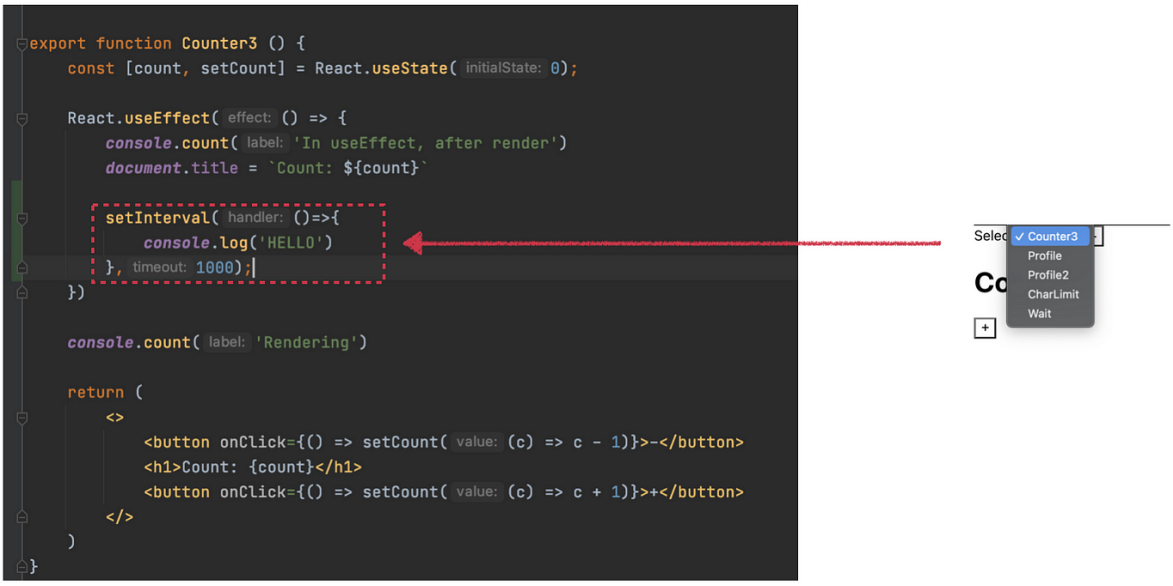


We've tied the useEffect call to the change of certain variables.

4.3. Clearing SideEffect Dependencies in useEffect

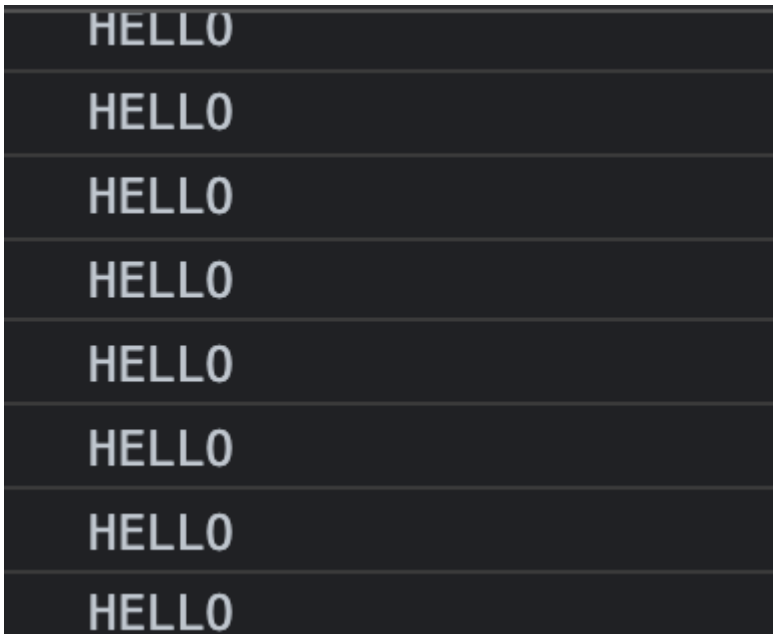
You have made some subscriptions in useEffect. For example, you want to listen to the event; you set up a timer, etc. But even if the component is removed from the screen, these dependencies will remain; you need to remove these dependencies.

Let's change our Counter example a little more and print HELLO on the screen every 1 second with the setInterval code ... And we don't want setInterval to run when we remove the component from the screen ...



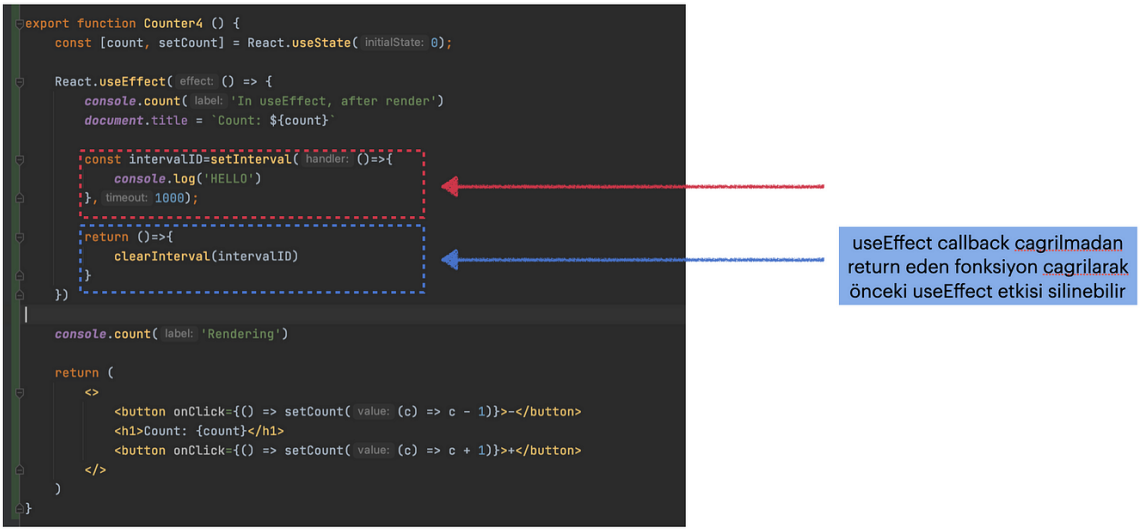
In this case, when you switch to rendering another component, does HELLO continue when this component is removed from the screen?

Yes. A situation occurred that we did not want. We are not rendering the component but the effect continues. This is undesirable for other components and the application.

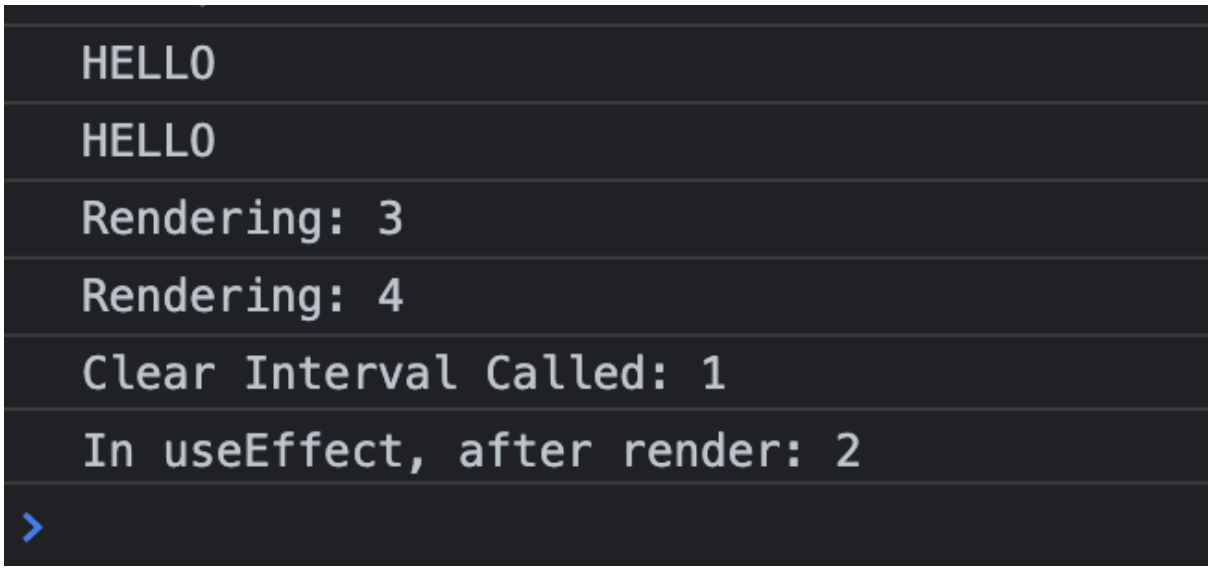


the component continues to say HELLO even when not rendered

In the Counter4 Component useEffect, when a **function return** is returned, this function is called before the useEffect callback is called, and you can remove the subscription that occurs in the side effect in this section.



When the code above runs, this component allows you to unsubscribe due to rendering or changing the entity to which the subscription has been added. This way, you can work without leaving a trace 😃



4.4 How to manage different props dependencies and their side effects with useEffect

Within the React component, useEffect can be used more than once, and different business logic algorithms can be written for different props. This is a very important requirement and usage pattern. Otherwise, it is not possible to manage all business logic in one useEffect.

React Hooks do not have the component lifecycle API in **React Class Component**s, so how can we perform some business logic by comparing the previous and next props in componentWillProps or componentDidUpdate props changes with React Hooks?

We need two things during useEffect.

- Props before and after
- Only be notified when your Props change.

This approach supports

- https://usehooks.com/usePrevious/. Allows you to reach the previous value of props and state.
- Or you need to build a state mechanic of that Props yourself.

In this article, when we write more than one useEffect that evaluates different states with useEffect and when we update different props such as (name, age, and height), we need a structure that will allow us to handle and evaluate them all differently.

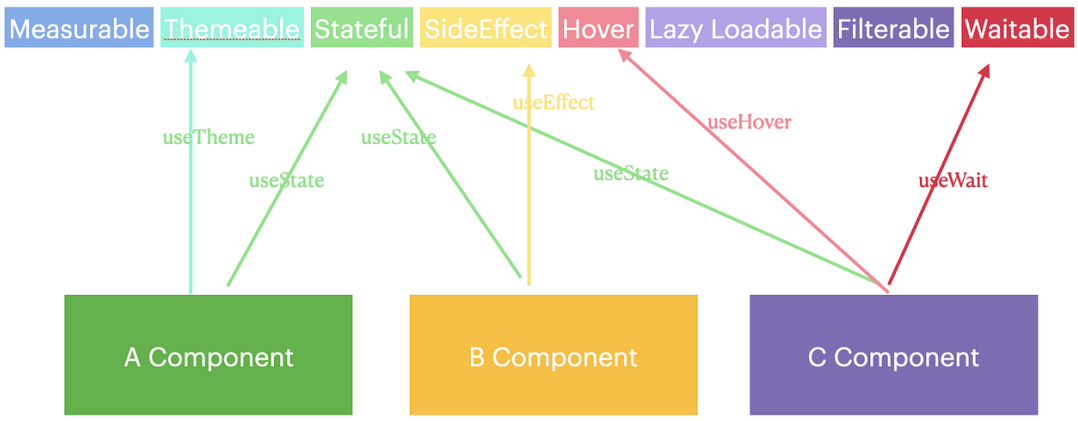
```
function User(props) {

    useEffect(() => {
        console.log('No filtering every render called')
    })

    useEffect(() => {
        console.log('called once at first')
    },[])

    useEffect(() => {
        console.log('useEffect Name updated:' + props.name);
    }, [props.name])

    useEffect(() => {
        console.log('useEffect Age updated:' + props.age)
    }, [props.age])

    useEffect(() => {
        console.log('useEffect Height updated:' + props.height)
    }, [props.height])

    return (
        <div>
            My
```

You can try how the above code works with this simple application https://onurdayibasi.dev/hooks-useeffect-01.

You can update the name, age, and height under Controls to see which **useEffect functions** they fall into.

5. What is a useContext?

I have explained that React Hook is a common set of behaviors for the components by hooking them, just like in the picture below.



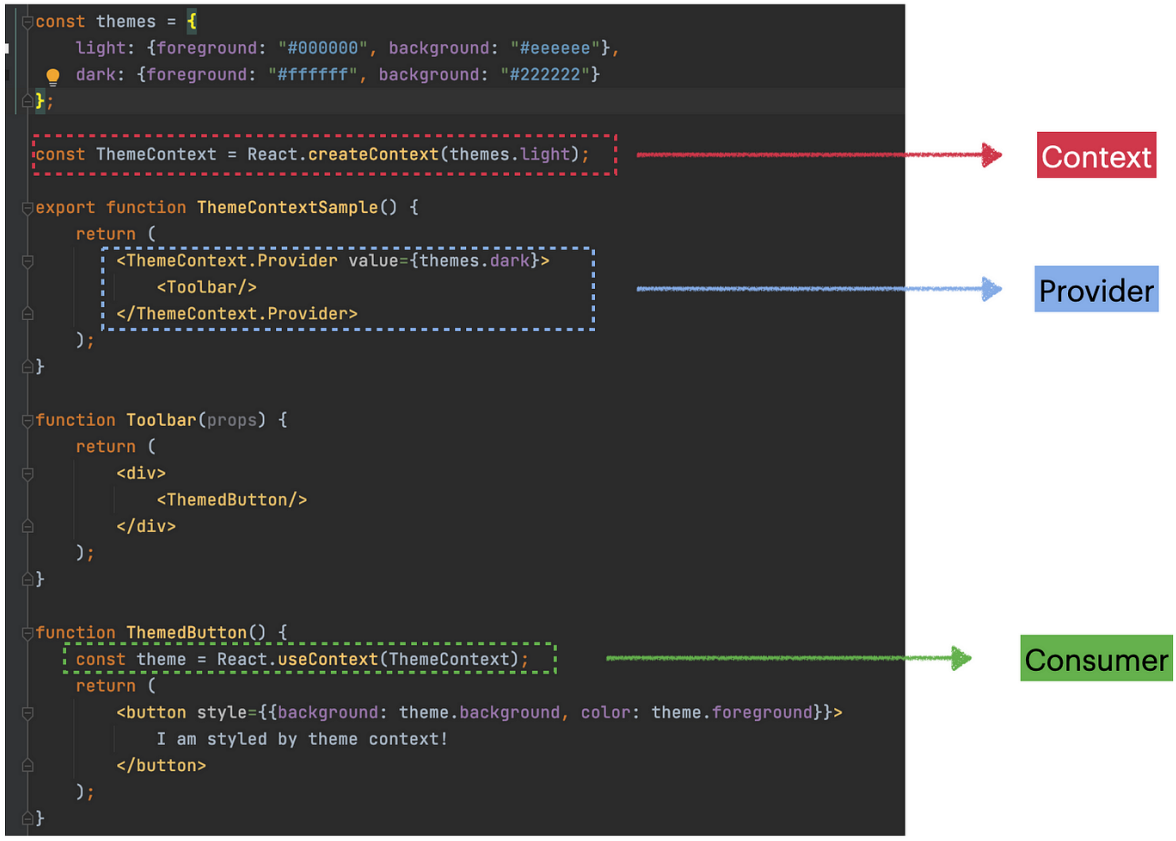
Aslında Context kavramından daha önceki **DOM Tree Problems** yazımızın içerisinde Virtual DOM ağaç düğümleri içerisinden bir düğümdeki bir çok bileşen düğümüne ortak veriyi bağlam verisini geçirmek için kullanılan bir API'dir. Bir çok

kütüphane Redux, Apollo Client, ReactRouter vb Provider Pattern altında Context API kullanarak işlerini halleder.

React Context kullanımını ayrıca bu linkten daha detaylı inceleyebilirsiniz. It basically consists of 3 parts;

- **Context:** Holds the values in the context.
- **Provider:** Provides the values in the context.
- **Consumer:** Makes the values in the context accessible from any node.

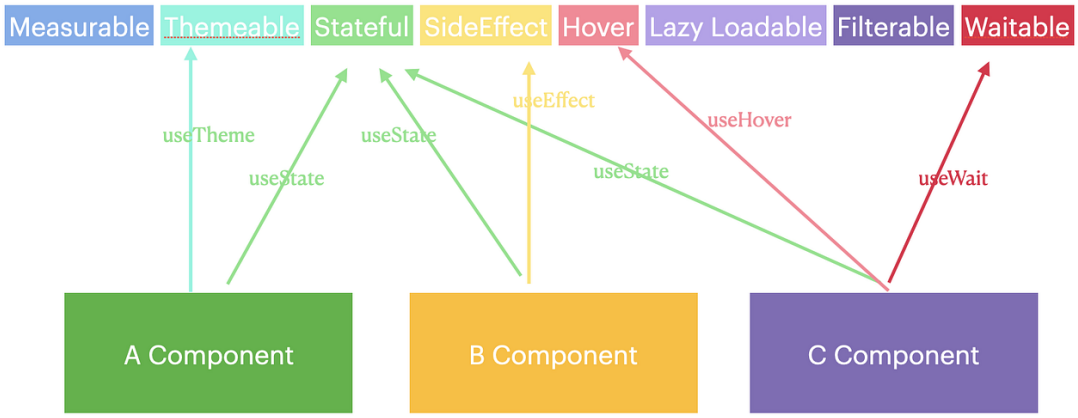
We also use the **useContext** function to use Class Component Consumer from Hook structures. This function makes the code more readable and allows it to be used in Functional Components.



https://tr.reactjs.org/docs/hooks-reference.html#usecontext

7. What is UseReducer?

I've explained that React Hook provides some common behaviors for components by hooking them, just like in the picture below.



useReducer provides developers with convenience at this stage if the components need a State Machine. If the state is managed as a flow, you can use the **useReducer** Hook instead of **useState**. For example, we have a counter value on our screen in the example below. There are UI Components that affect it. The simplest method is increasing it by 1, decreasing it by 1, or setting it to 0 using useState.

0

+ - Reset

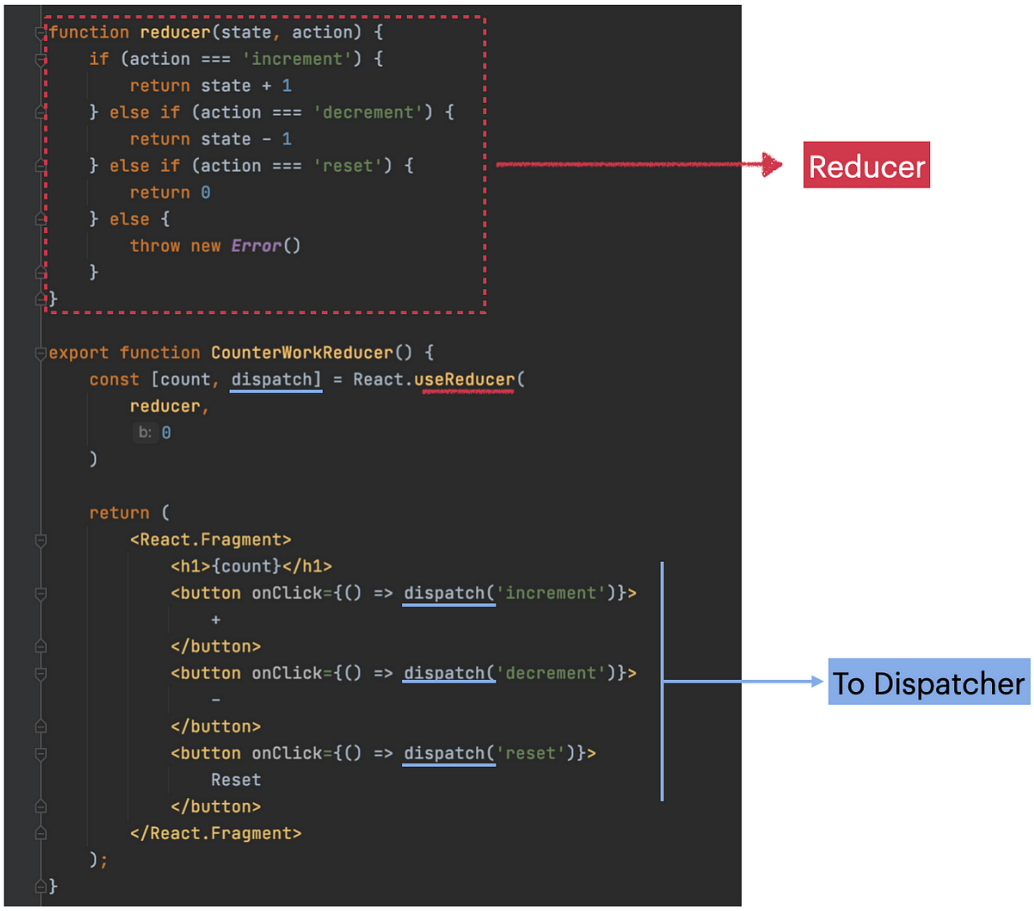
However, useReducer allows developers to manage this and more complex state transitions with dispatch logic, similar to the Redux mechanism.

- The part in red below is the reducer function and the current state and this state is waiting for the command to transition to the next stage.
- In the 2nd part, we get useReducer dispatch and state(count) information
- The dispatch function transmits the "string" command as the relevant reducer (state, action).
- Then our component will be rendered again.

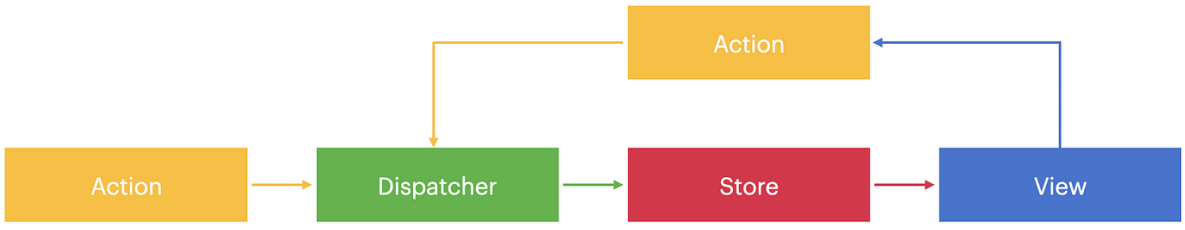
https://reactjs.org/docs/hooks-reference.html#usereducer

Sometimes I get questions like this. Is Redux, along with React Hook **useReducer and useContext**, over? Do we need to use Redux? Why do they always ask this question?

Because the first installation phase of Redux is **complicated** to learn, while Redux is an App level event, useReducer is a component-level Hook. They are similar but not interchangeable technologies.



https://reactjs.org/docs/hooks-reference.html#usereducer



Flux Architecture

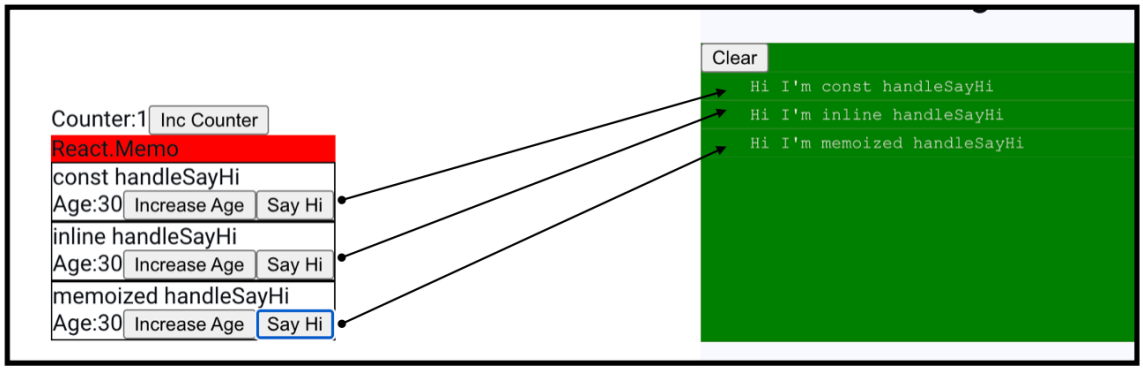
, we keep a structure similar to Redux, our Global variables like **Context** → Store. You can provide the Flux pattern by calling it through the reducer function and

dispatch function you will give useReducer. But as I said, it is valid for simple examples and uses.

6. What is useCallback?

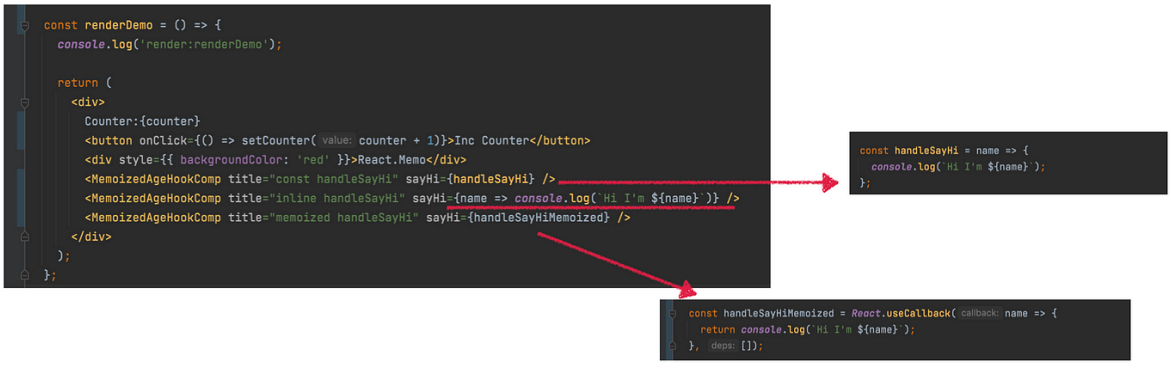
There were methods we could use on Class Component (Class Components) via **state, and props** change (forceUpdate, shouldComponentDidUpdate). Here we will focus on using **useCallback and useMemo** in React Hooks.

Let's change the logic in the first scenario. We did memoization with the same Hook component React.memo. But this time, we pass a callback function in it. (Example application)

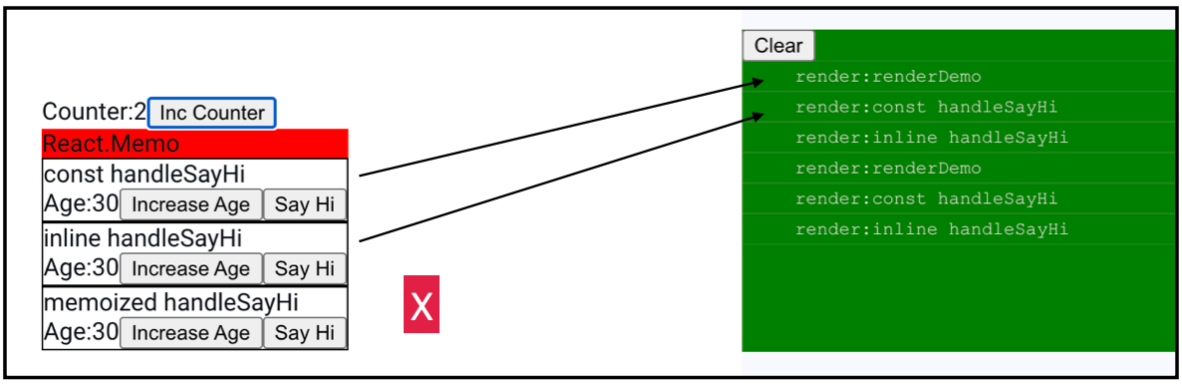


We perform three different types of callback passing.

- Defining ***const*** callback and passing it to the child component as props
- Stretching props callback function to component as ***inline***
- and finally passing the function to the component using ***useCallback***...



Let's increase the Counter in the Parent component and see which ones render.



Although we use React.memo, the update in the parent component has started to trigger the child components; only when we cover the callback with **useCallback**, the child component did not render itself. So why?

The reason is React.memo if we do not give any control check operation if we do not make the following props comparison,

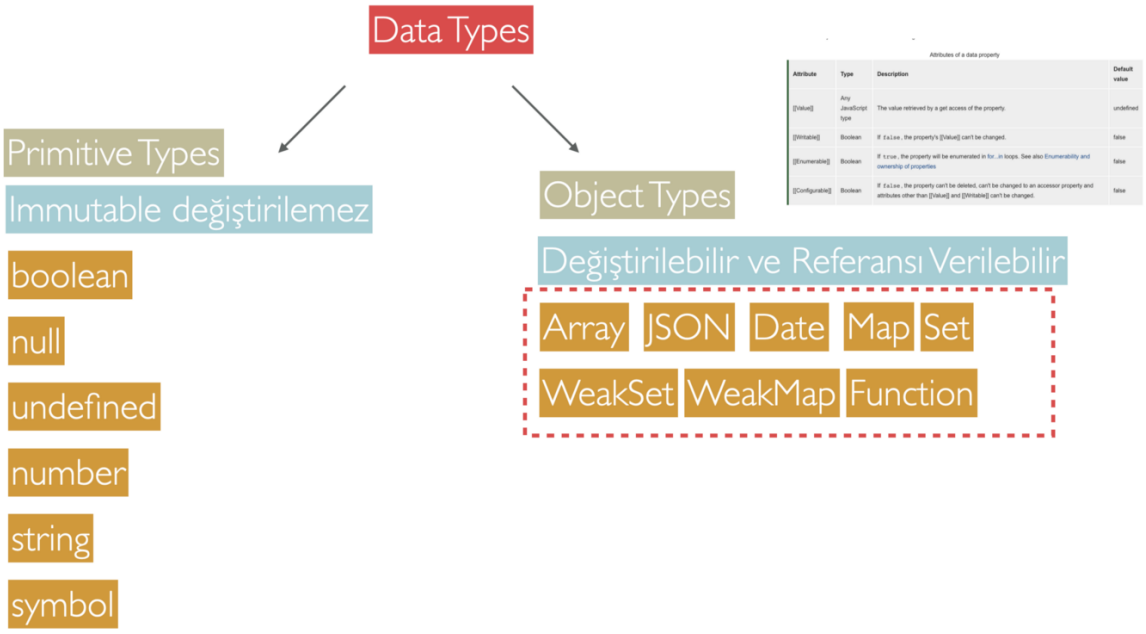
```
function MyComponent(props) {
  /* render using props */
}
function areEqual(prevProps, nextProps) {
  /*
  return true if passing nextProps to render would return
  the same result as passing prevProps to render,
  otherwise return false
  */
}
export default React.memo(MyComponent, areEqual);
```

https://reactjs.org/docs/react-api.html#reactmemo

React shallowly compares it, meaning it compares props parameters with ===.

> By default, it will only shallowly compare complex objects in the props object. If you want control over the comparison, you can also provide a custom comparison function as the second argument.

Well (===) I explained in JS Datatypes. This will work for primitive types, but for others, it will make a comparison by reference.



In this case, since we pass a **new callback** function to the component during each render, how can we make their references the same?

This is where the **useCallback** hook comes into play. As long as [a,b] in the dependency array in this callback does not change, it is ensured to use the same reference.

useCallback

```
const memoizedCallback = useCallback(
  () => {
    doSomething(a, b);
  },
  [a, b],
);
```

https://reactjs.org/docs/hooks-reference.html#usecallback

7. What is useMemo?

Many methods are used to achieve performance other than the normal rendering logic of React components. In this article, we will talk about useMemo.

In the topics we have explained so far, our goal has always been to prevent component re-rendering. To summarize

- With ClassComponent **shoudComponentUpdate,** we could prevent unnecessary rendering of the component.
- With **PureComponent** or **React.memo,** we could prevent unnecessary rendering of the component. With **useCallback**, we ensured that the callback references were the same and React.memo worked effectively.

As you can see, the abovementioned approaches aim to prevent the component from being drawn repeatedly.

How can we prevent the functions that return a value in a component such as calculation, IO, etc.**** from being executed repeatedly if they take the same parameters? Answer → ***useMemo***

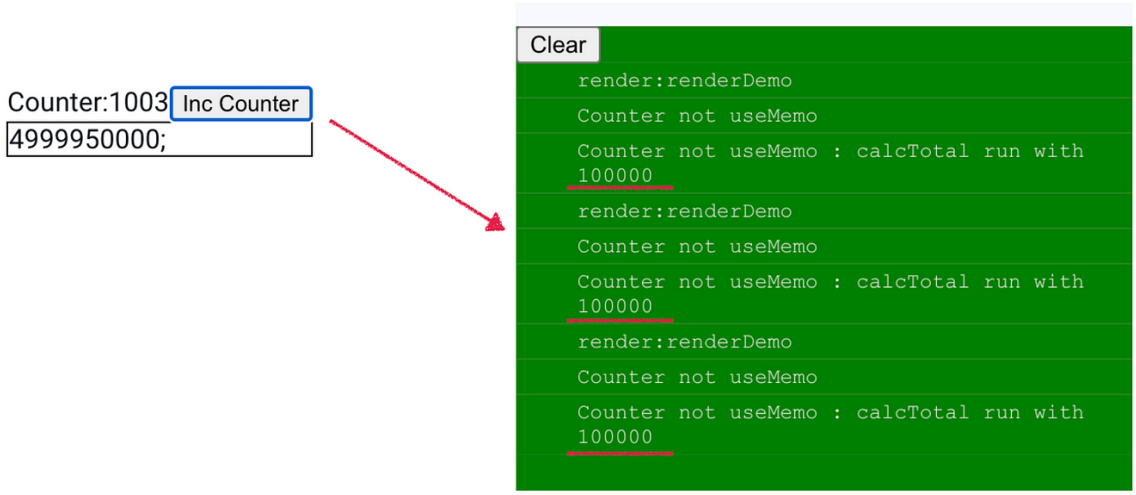
https://reactjs.org/docs/hooks-reference.html#usememo

useMemo is simply a hook that allows a function that returns a value to be called with the same parameters, returns the cached value, and prevents re-calling if the previously called values are the same.

Instead of controlling the rendering of the component below, sometimes it makes more sense to control the function. Imagine if we memoize the computeExtensiveValue function as shown in the image below. This function will be executed in cases where it is not needed to be executed repeatedly.

```
function Counter(props) {
  console.log(`Counter not useMemo`);

  const computeExpensiveValue = maxNumber => {
    console.log('Counter not useMemo : calcTotal run');
    let total = 0;
    for (let i = 0; i < maxNumber; i++) {
      total += i;
    }
    return total;
  };

  const result = computeExpensiveValue( maxNumber: 100000);

  return (
    <div style={{ border: '1px solid black' }}>
      {props.title}
      {result};
      <br />
    </div>
  );
}
```

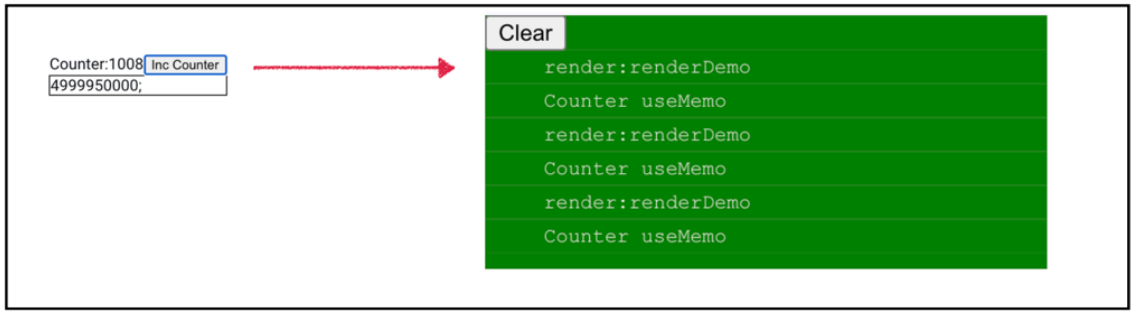
When we run the code below, the function that obtains computeExpensiveValue is called repeatedly.



This time let's memoize the function with useMemo, so that the generated value is stored somewhere.

```
//===========
function CounterUseMemo(props) {
  console.log(`Counter useMemo `);

  const computeExpensiveValue = maxNumber => {
    console.log('Counter useMemo : calcTotal run');
    let total = 0;
    for (let i = 0; i < maxNumber; i++) {
      total += i;
    }
    return total;
  };

  const result = useMemo( factory: () => computeExpensiveValue( maxNumber: 100000), deps: []);

  return (
    <div style={{ border: '1px solid black' }}>
      {props.title}
      {result};
      <br />
    </div>
  );
}
```

And you can see that computExtensiveValue is not called over and over again with the method we obtained.



8. What is useRef?

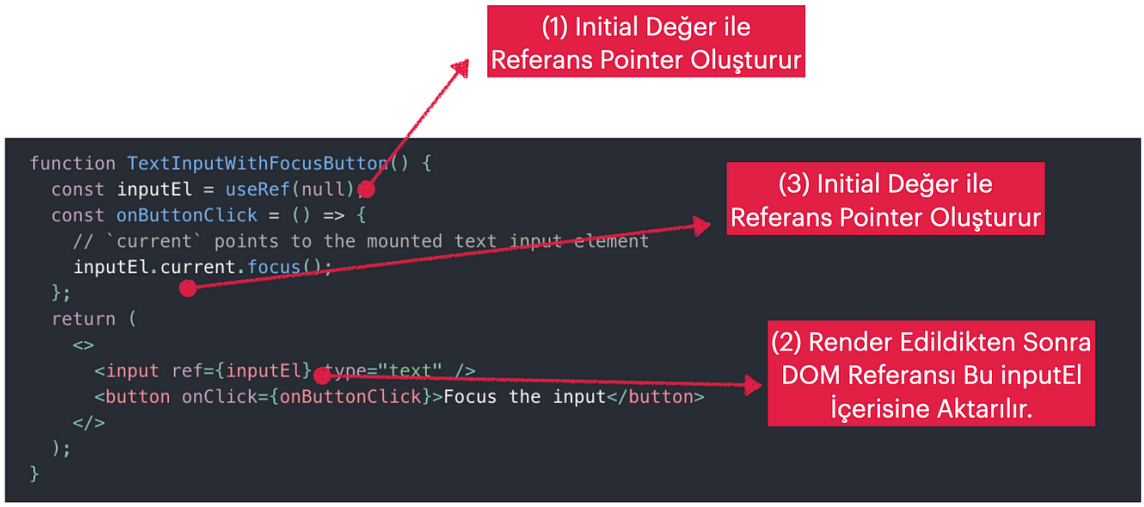
A Ref object is a Hook that we use to access the link of the component in the VirtualDOM to the actual DOM reference in the browser.

```
const refContainer =useRef(initialValue);
```

useRef returns us a reference object. This object is our

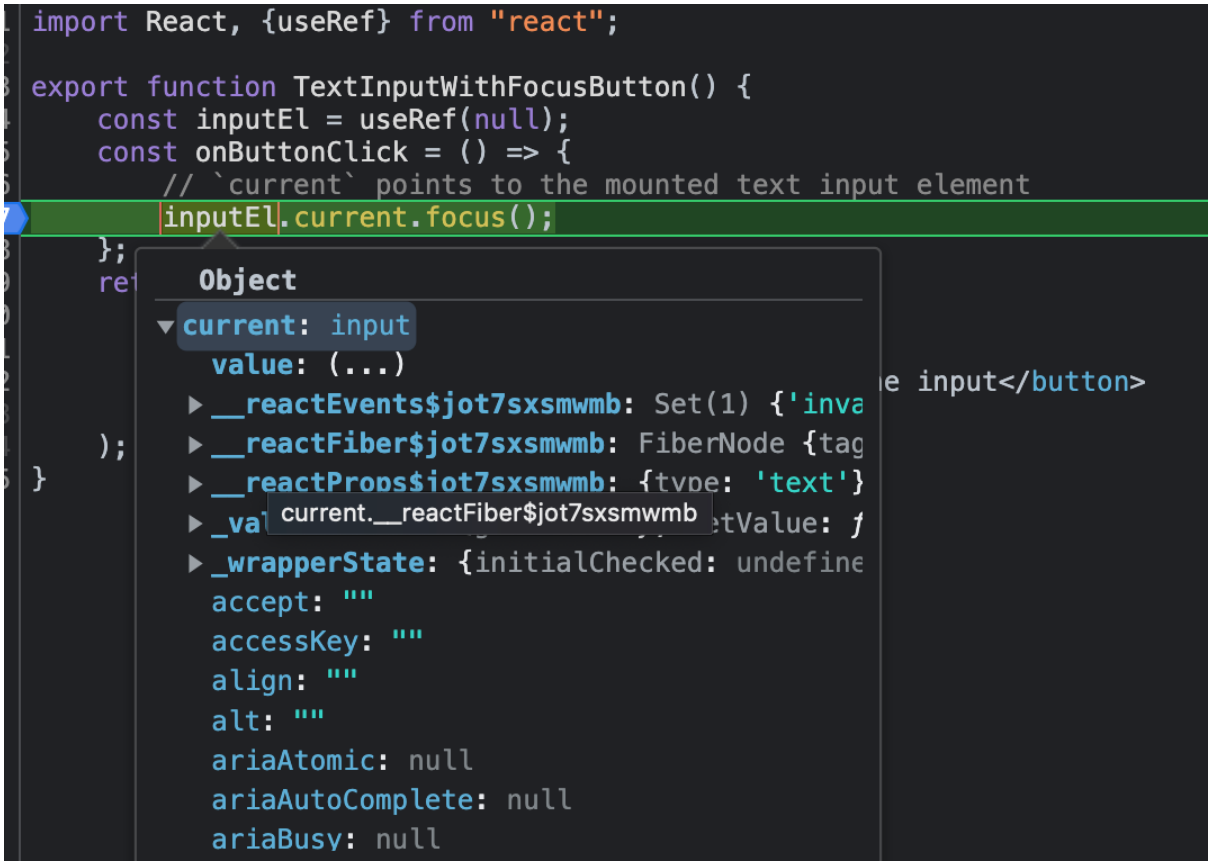
> useRef returns a mutable ref object whose .current property is initialized to the passed argument (initialValue). The returned object will persist for the full lifetime of the component.

In the **1st step,** we specify that we will use a Ref Object with null.



In the **2nd step,** we assign the ref to the input vb component. We can now access this DOM reference via current in the VirtualDOM mapping.

In the 3rd step, we can now operate on the actual DOM element when we click on a button.



inputEl element's ref value

9. What is useImperativeHandle?

This Hook function provides API for imperative access to your component from outside, allowing the parent component to consume the API in this component.

Here you need to use ***useImperativeHandle*** along with ***useRef*** and ***forwardRef***.

In this scenario, we open the FancyInput component to provide API to the outside. Here in **useImperativeHandle** we write the functions that we will provide to the parent component. For example **myFocus, d** then we make its reference available to the parent component with ***forwardRef***.

10. **What is **useLayoutEffect?

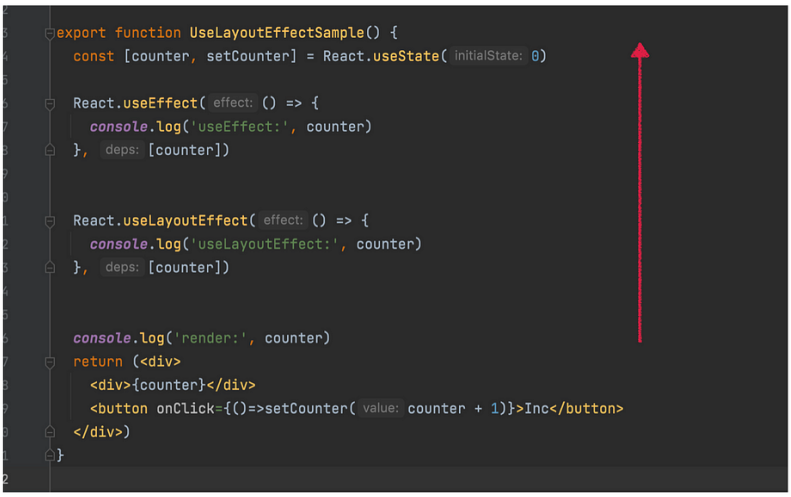
There is a ***useEffect*** function that is called after rendering with hooks. And with ***useEffect***, we can reflect the effect of changes such as Prop, State, etc., that cause a change for the component as a sideEffect

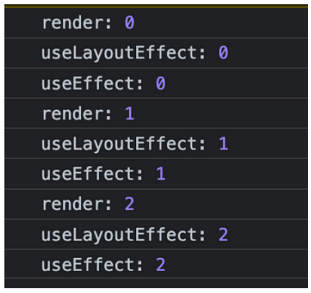
So then, why do we need ***useLayoutEffect***. It is called synchronously immediately before the paint operation after the DOM Mutation occurs. So it informs us developers about DOM Mutation before paint.

```
Render → React Updates DOM →useLayoutEffect→ BrowserPaint Screen → useEffect
```

Class Component useEffect, the useLayoutEffect call actually corresponds to the **componentDidMount** or **componentDidUpdate** callback calls.

useLayoutEffect is used to get the actual information about the dimensions of the relevant components and then operate on them according to the DOM Mutation without the screen paint operation.





Using useEffect and useLayoutEffect

11. **What is useDebugValue?

We are continuing with Hook. In this article, we will focus on the useLayoutEffect Hook in React.

You have written your own Custom Hook and want to show information about this Custom Hook component in React DevTools. You can do this simply with the following command.

```
useDebugValue(value)
```

For example, let's write ourselves a custom Hover Hook like useHover if we print the current status with useDebugValue in it.

```
import React from 'react';
export function useHover () {
    const [hovering, setHovering] = React.useState( initialState: false)

    const mouseOver = () => setHovering( value: true)
    const mouseOut = () => setHovering( value: false)

    const handlers = {
        onMouseOver: mouseOver,
        onMouseOut: mouseOut
    }

    // Show a label in DevTools next to this Hook
    React.useDebugValue( value: hovering ? 'Hovering OK' : 'Hovering Not OK');
    return [hovering, handlers]
}
```

useHover Custom Hook

For example, when we Hover over the Blue Component, we can show its state as a property under hooks in React DevTools.