Hello,

KDT 웹 개발자 양성 프로젝트

1기! 39th

with

Props

MainHeader 에 props 적용

- MainHeader 의 매개변수 전달 부분에 props 를 추가하고 사용할 props 명을 적어 주시면 됩니다!

```
import React from "react";

function MainHeader(props) {
    return (
        <h1>{props.text}</h1>
    )
}

export default MainHeader;
```

src/components/MainHeader.js

MainHeader 에 props 적용

- MainHeader 의 인자 전달 부분에 {} 를 추가하여 인자 전달 부분에서 구조 분해 할당으로 props 를 받아와서 사용해도 됩니다!

```
import React from "react";

function MainHeader({ text }) {
    return (
        <h1>{text}</h1>
    )
}

export default MainHeader;
```

src/components/MainHeader.js

```jsx
function App() {
  return (
    <div className="App">
      <MainHeader text="Hello, props world!" />
      <MainHeader text="Bye Bye, props world!" />
      <MainHeader text="Well come back, props world!" />
    </div>
  );
}

export default App;
```

src/App.js

Hello, props world!

Bye Bye, props world!

Well come back, props world!

Props

활용하기

배열을 전달하고 props 로 받아서 처리!

- Props 로는 배열 같은 다양한 자료형의 전달이 가능합니다!
- 배열을 받아서 처리하는 CustomList.js 컴포넌트를 만들어 봅시다!

```
function CustomList(props) {
    return (
        <ul>
            {props.arr.map((el) => {
                return <li>{el}</li>
            })}
        </ul>
    )
}

export default CustomList;
```

src/components/CustomList.js

App.js 에서 배열을 전달

- App 에서 임의의 배열을 만들어서 전달하기!

```
function App() {
  const nameArr = ['뽀로로', '루피', '크롱이'];
  return (
    <div className="App">
      <CustomList arr={nameArr} />
    </div>
  );
}

export default App;
```

src/App.js

뽀로로
루피
크롱이

객체를 전달하고 props 로 받아서 처리!

- 객체을 받아서 처리하는 CustomObj.js 컴포넌트를 만들어 봅시다!

```
function CustomObj(props) {
    const { name, age, nickName } = props.obj;
    return (
        <div>
            <h1>이름 : {name}</h1>
            <h2>나이 : {age}</h2>
            <h3>별명 : {nickName}</h3>
        </div>
    )
}

export default CustomObj;
```

src/components/CustomObj.js

App.js 에서 객체를 전달

- App 에서 임의의 객체을 만들어서 전달하기!

```
function App() {
  const pororoObj = {
    name: "뽀로로",
    age: "5",
    nickName: "사고뭉치"
  }
  return (
    <div className="App">
      <CustomObj obj={pororoObj} />
    </div>
  );
}
```

src/App.js

이름 : 뽀로로

나이 : 5

별명 : 사고뭉치

컴포넌트 꾸미기

인라인으로 꾸미기!

src/style/TestCss.css

```
const divStyle = {
    backgroundColor: "orange"
}

const headingStyle = {
    color: "blue"
}

const spanStyle = {
    backgroundColor: "pink",
    fontWeight: "700"
}

export default function TestCss() {
    return (
        <div className="component-root" style={divStyle}>
            <h1 style={headingStyle}>CSS 테스트 컴포넌트 입니다</h1>
            <span style={spanStyle}>해당 컴포넌트를 기본 CSS로 꾸며 보아요!</span>
        </div>
    )
}
```

CSS 테스트 컴포넌트 입니다

해당 컴포넌트를 기본 CSS로 꾸며 보아요!

기본 CSS로 꾸미기

```css
div.component-root {
    background-color: orange;
}

div.component-root h1 {
    color: red;
}

div.component-root span {
    background-color: white;
    font-weight: 700;
}
```

src/style/TestCss.css

```jsx
import '../style/TestCss.css';

export default function TestCss() {
    return (
        <div className="component-root">
            <h1>CSS 테스트 컴포넌트 입니다</h1>
            <span>해당 컴포넌트를 기본 CSS로 꾸며 보아요!</span>
        </div>
    )
}
```

src/components/TestCss.js

CSS 테스트 컴포넌트 입니다

해당 컴포넌트를 기본 CSS로 꾸며 보아요!

Sass 사용하기!

SASS / SCSS 로 꾸미기!

- Src/style 폴더에 TestCss.scss 파일 생성

```
div.component-root {
    background-color: skyblue;
    & > h1 {
       color: dodgerblue;
    }

    & > span {
        background-color: pink;
        font-weight: 700;
    }
}
```

src/style/TestCss.scss

```jsx
import '../style/TestCss.scss';

export default function TestCss() {
    return (
        <div className="component-root">
            <h1>CSS 테스트 컴포넌트 입니다</h1>
            <span>해당 컴포넌트를 기본 CSS로 꾸며 보아요!</span>
        </div>
    )
}
```

src/components/TestCss.js

CSS 테스트 컴포넌트 입니다

해당 컴포넌트를 기본 CSS로 꾸며 보아요!

Styled Components


styled
components

Styled Components 사용하기

- Styled Components 는 독특하게 사용이 됩니다!

```
export default function TestStyled() {
    return (
        <MyDiv>
            <MyHeading>h1 태그 입니다!</MyHeading>
            <MySpan>span 태그 입니다!</MySpan>
        </MyDiv>
    )
}
```

src/components/TestStyled.js

- 먼저 자기만의 이름으로 태그를 구성 합니다!

Styled Components 사용하기

- 각각의 태그를 변수에 할당하고 해당 태그의 실제적인 태그명을 styled 를 이용하여 지정합니다!

```
const MyDiv = styled.div`
    background-color: orange;
`;

const MyHeading = styled.h1`
    color: blue;
`

const MySpan = styled.span`
    background-color: pink;
    font-weight: 700;
`
```

src/components/TestStyled.js

조건부 렌더링!

```jsx
import ConditionalRender from './components/ConditionalRender';
import { useState } from 'react';

function App() {

  const [condition, setCondition] = useState("보이기");

  const onChange = () => {
    condition === "보이기" ? setCondition("감추기") : setCondition("보이기");
  }

  return (
    <div className="App">
      {condition === "감추기" && <ConditionalRender />}
      <button onClick={onChange}>{condition}</button>
    </div>
  );
}

export default App;
```

src/App.js

```
function App() {

  const [condition, setCondition] = useState("1번");
  const onChange = () => {
    condition === "1번" ? setCondition("2번") : setCondition("1번");
  }



  return (
    <div className="App">
      {condition === "1번" ? <PracticeOne text={condition} /> : <PracticeTwo text={condition}
/>}
      <button onClick={onChange}>{condition}</button>
    </div>
  );
}

export default App;
```

useRef

```jsx
import { useState, useRef } from "react";

export default function TestRef() {
    const [text, setText] = useState("안녕하세요!");

    const inputValue = useRef();

    const onChangeText = () => {
        setText(inputValue.current.value);
    }

    return (
        <div>
            <h1>{text}</h1>
            <input ref={inputValue} onChange={onChangeText}></input>
        </div>
    )
}
```

src/components/TestRef.js

useRef

focus

```jsx
import { useState, useRef } from "react";

export default function TestRef() {
    const input1 = useRef();
    const input2 = useRef();

    const changeFocusOne = () => {
        input1.current.focus();
    }

    const changeFocusTwo = () => {
        input2.current.focus();
    }

    return (
        <div>
            <input ref={input1}></input>
            <input ref={input2}></input>
            <br></br>
            <button onClick={changeFocusOne}>1</button>
            <button onClick={changeFocusTwo}>2</button>
        </div>
    )
}
```

src/components/ChangeFocus.js

HOOKS

태초의 리액트는…

- 태초의 리액트는 클래스형 컴포넌트만 사용 했습니다
- 그리고 이전에 배우셨던 컴포넌트 별 상태 관리 및 라이프 사이클 기능을 지원 했습니다!
- 하지만 이것들이 불편하다 느껴졌고, 리액트는 함수형 컴포넌트가 더 편리하고 효율적인 것을 알게 되어서 함수형 컴포넌트로 갈아 탔죠!

그래서 탄생, React HOOKS

- 다만 기존 클래스형 컴포넌트에서 사용하던 편리한 기능(리액트의 핵심)을 함수형에 적용하려니 기존 것을 그대로 사용할 수는 없었고 새로운 것이 필요 했습니다
- 그래서 탄생한 것이 React HOOKS 입니다
- 앞에 use 가 붙은 애들이 HOOKS 들이죠
  - useState / useRef / useEffect / useContext / useMemo / useCallback / useReducer

HOOKS

React.Fragment

React.Fragment?

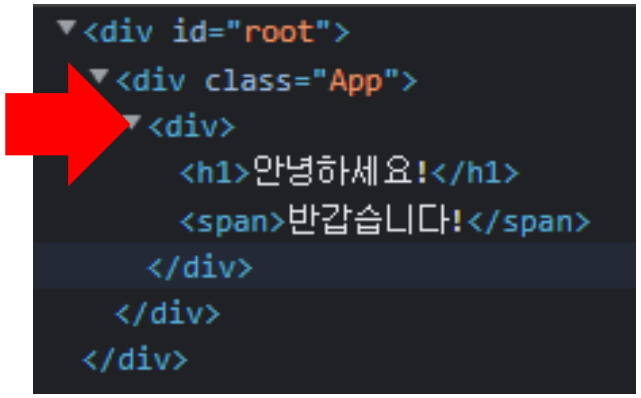
- 자, 정말 간단한 컴포넌트를 만들어 볼게요!

```
export default function ReactFragment() {
  return (
    <div>
      <h1>안녕하세요!</h1>
      <span>반갑습니다!</span>
    </div>
  );
}
```

src/components/ReactFragment.js

React.Fragment?

- 그리고 페이지에서 개발자 도구를 열어 보면!



- 간단한 컴포넌트임에도 div 요소가 하나 추가 되었네요!
- 그럼, 저 div 를 없앨수 없을까요?

React.Fragment?

- 리턴 값에서 최상위 태그 역할을 하는 DIV를 빼보시죠!

```jsx
export default function ReactFragment() {
  return (
    <h1>안녕하세요!</h1>
    <span>반갑습니다!</span>
  );
}
```

```
ERROR in [eslint]
src\components\ReactFragment.js
  Line 4:6:  Parsing error: Adjacent JSX elements must be wrapped in an enclosing tag. Did you want a JSX fragment <>...</>? (4:6)
```

- 바로 에러가 뜹니다! 그런데 React 가 JSX fragment 를 추천하네요!?

React.Fragment!

- 실제 리액트에서 컴포넌트를 조합할 때, 최상위에 div 를 사용하지 않고 반환해야만 하는 경우가 생기게 됩니다!
- CSS 가 깨진다거나, 테이블 요소 사이에 div 요소가 들어가면 에러가 뜨기 때문이죠!
- 그럴 때 쓰는 것이 바로 React.Fragment 입니다!

React.Fragment!

- 이제 컴포넌트를 React.Fragment 로 감싸 봅시다!
- 이건 React 라이브러리의 기능이므로 React 라이브러리 추가 필요

```jsx
import React from "react";

export default function ReactFragment() {
  return (
    <React.Fragment>
      <h1>안녕하세요!</h1>
      <span>반갑습니다!</span>
    </React.Fragment>
  );
}
```

src/components/ReactFragment.js

React.Fragment!

- 이제는 div 가 사라졌네요!?

```
▼<div id="root">
  ▼<div class="App">
      <h1>안녕하세요!</h1>
      <span>반갑습니다!</span>
    </div>
  </div>
```

- 바로 이러한 역할을 해주는 것이 React.Fragment 입니다!

<> </>

- 개발자들은 축약의 민족이기 때문이 이렇게 긴 코드를 용납 못합니다!
- <React.Fragment> 는 <> 로 대체가 가능합니다! :)

```jsx
import React from "react";

export default function ReactFragment() {
  return (
    <>
      <h1>안녕하세요!</h1>
      <span>반갑습니다!</span>
    </>
  );
}
```

src/components/ReactFragment.js

React.Fragment

가 필요할 때!

CSS 님이 화가 나셨어!

- 먼저 App.js 에 가서 ReactFragment 컴포넌트와 동일한 코드를 작성해 봅시다

```
function App() {
  return (
    <div className="App">
      <h1>안녕하세요</h1>
      <span>반갑습니다</span>
    </div>
  );
}

export default App;
```

src/App.js

CSS 님이 화가 나셨어!

- 그리고 App.css 에 아래의 코드도 추가해 봅시다!

```css
.App {
  display: flex;
  justify-content: space-between;
}

.App h1 {
  background-color: skyblue;
}

.App span {
  background-color: orange;
}
```

src/App.css

결과물 화면~!

안녕하세요

반갑습니다

- 이런 결과물 화면이 나와야 정상입니다!

ReactFragment 컴포넌트를 적용!

- ReactFragment 컴포넌트의 최상위 요소를 div 로 변경해서 적용시켜 봅시다!

```
import React from "react";

export default function ReactFragment() {
  return (
    <div>
      <h1>안녕하세요!</h1>
      <span>반갑습니다!</span>
    </div>
  );
}
```

src/components/ReactFragment.js

ReactFragment 컴포넌트를 적용!

안녕하세요!

반갑습니다!

- Div 가 생기게 되어서 이전 결과물과는 완전히 다른 결과물이 나오게 됩니다!
- 따라서, 이런 일을 피하려면 React.Fragment 를 쓰셔야 합니다!

나는 너에게 속할 수 없어!

- 테이블 요소에 테이블 내용을 컴포넌트로 삽입하는 경우를 생각해 봅시다
  (사실 이게 쓸 일이 거의 없긴 합니다 ㅎㅎㅎ;;)
- 그런데 테이블 요소 안에는 div 태그가 들어가지 못합니다!
- 이럴때 문제가 생기는데 이것을 React.Fragment 로 해결이 가능합니다!

```jsx
import TableColumn from
"./components/TableColumn";

function App() {
  return (
    <div className="App">
      <table border="1">
        <tbody>
          <tr border="1">
            <td>1</td>
            <td>2</td>
            <td>3</td>
          </tr>
          <tr>
            <TableColumn />
          </tr>
        </tbody>
      </table>
    </div>
  );
}

export default App;
```

src/App.css

```jsx
import React from "react";

export default function ReactFragment() {
  return (
    <div>
      <td>a</td>
      <td>b</td>
      <td>c</td>
    </div>
  );
}
```

src/components/TableColumn.js

```
Download the React DevTools for a better development experience: https://reactjs
▸Warning: validateDOMNesting(...): <td> cannot appear as a child of <div>.
    at td
    at div
    at TableColumn
    at tr
    at tbody
    at table
    at div
    at App
▸Warning: validateDOMNesting(...): <div> cannot appear as a child of <tr>.
    at div
    at TableColumn
    at tr
    at tbody
    at table
    at div
    at App
>
```

useState

useRef

Variable

3가지 타입에 대해서 정리해 봅시다!

- 지금까지 배운 useEffect 의 state 그리고 useRef 의 ref 와 리액트 내부의 변수가 렌더링에 따라 어떤 식으로 변화하는지 알아 봅시다!

State

변경

Ref

변경

Variable

변경

?

코드로 봅시다!

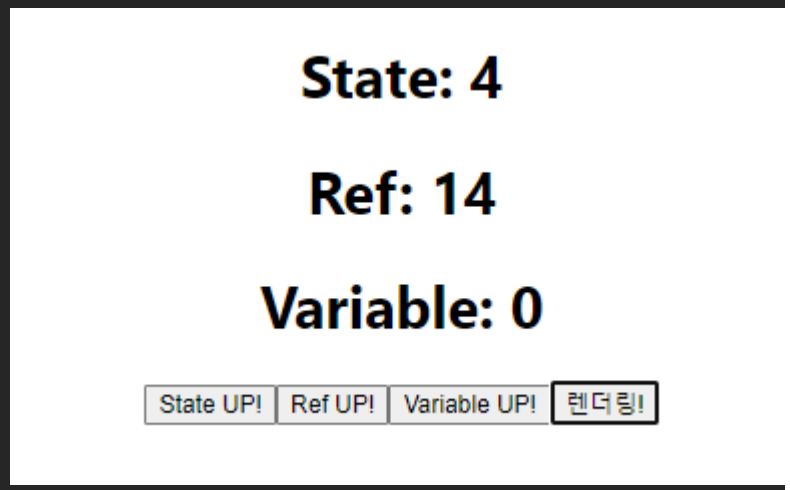
- 세 가지 버튼으로 각각 state / ref / variable 값을 올리는 컴포넌트를 구성해 봅시다!
- 그리고 값의 변화 없이 컴포넌트 리렌더링을 위해서 하나의 버튼을 더 만들어 봅시다
- 그리고 각각의 버튼을 클릭하면서 해당 값의 변화에 대해 관찰해 봅시다!

```jsx
import { useRef, useState } from "react";

const Comparing = () => {
    const [countState, setState] = useState(0);
    const [render, setRender] = useState(0);
    const countRef = useRef(0);
    let countVar = 0;

    const countUpState = () => {
        setState(countState + 1);
        console.log('State: ', countState);
    }

    const countUpRef = () => {
        countRef.current = countRef.current + 1;
        console.log('Ref: ', countRef.current);
    }

    const countUpVar = () => {
        countVar = countVar + 1;
        console.log('Variable: ', countVar);
    }

    const reRender = () => {
        setRender(render + 1);
    }
```

```jsx
    return (
        <>
            <h1>State: {countState}</h1>
            <h1>Ref: {countRef.current}</h1>
            <h1>Variable: {countVar}</h1>
            <button onClick={countUpState}>State UP!</bu
            <button onClick={countUpRef}>Ref UP!</button
            <button onClick={countUpVar}>Variable UP!</bu
            <button onClick={reRender}>렌더링!</button>
        </>
    )
}

export default Comparing;
```

src/components/Comparing.js

```jsx
import { useRef, useState } from "react";

const Comparing = () => {
    const [countState, setState] = useState(0);
    const [render, setRender] = useState(0);
    const countRef = useRef(0);
    let countVar = 0;

    const countUpState = () => {
        setState(countState + 1);
        console.log('State: ', countState);
    }

    const countUpRef = () => {
        countRef.current = countRef.current + 1;
        console.log('Ref: ', countRef.current);
    }

    const countUpVar = () => {
        countVar = countVar + 1;
        console.log('Variable: ', countVar);
    }

    const reRender = () => {
        setRender(render + 1);
    }

    return (
        <>
            <h1>State: {countState}</h1>
            <h1>Ref: {countRef.current}</h1>
            <h1>Variable: {countVar}</h1>
            <button onClick={countUpState}>State UP!</button>
            <button onClick={countUpRef}>Ref UP!</button>
            <button onClick={countUpVar}>Variable UP!</button>
            <button onClick={reRender}>렌더링!</button>
        </>
    )
}

export default Comparing;
```



src/components/Comparing.js

Quiz, 각각의 값의 변화!

- State 는 변경이 되면 바로 리렌더링이 일어나기 때문에 버튼을 클릭하면 바로바로 반영이 됩니다!
- Ref 는 리렌더링이 일어나지 않기 때문에 버튼을 클릭하면 Console.log 에만 찍히다가 컴포넌트가 리렌더링이 되면 그 때 한꺼번에 반영 됩니다
- 그런데 왜 변수는 Console.log 에는 잘 찍히다가 렌더링이 되면 0 만 나올까요?

Life Cycle

Mount

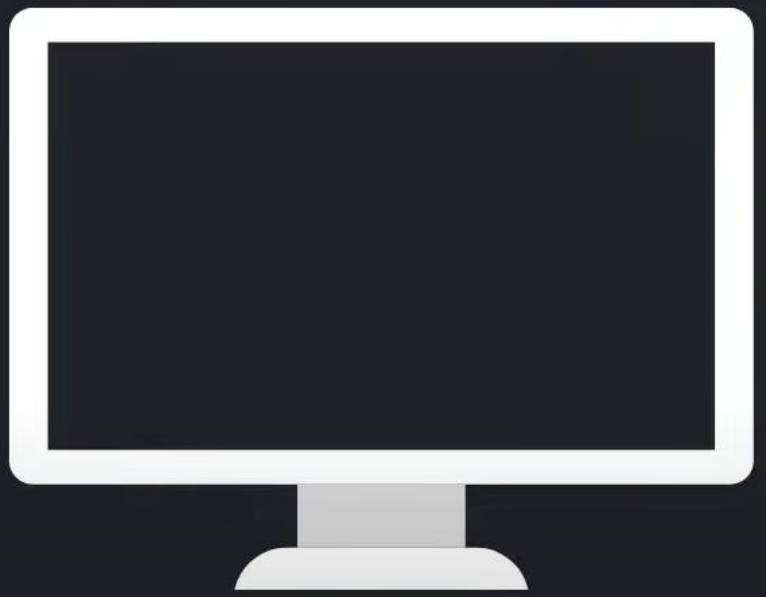
화면에 첫 렌더링

Update

다시 렌더링

Unmount

화면에서 사라질때



컴포넌트의 Life Cycle

- 컴포넌트는 최초에 화면에 등장 할 때 → Mount
- 컴포넌트의 state 변화로 리렌더링 될 때 → Update
- 화면에서 사라질 때 → Unmount
- 생명 주기를 가집니다!

클래스형 컴포넌트의 Life Cycle

- 리액트의 장점은 이러한 상황에서 컴포넌트별 상태 관리 및 리렌더링에 있기 때문에 리액트는 Life Cycle 에 대한 기능이 많습니다!
- 따라서 각각의 Life Cycle 상황에 맞게 특정 기능을 수행할 수 있도록 다양한 기능을 제공 했습니다

“Render Phase”

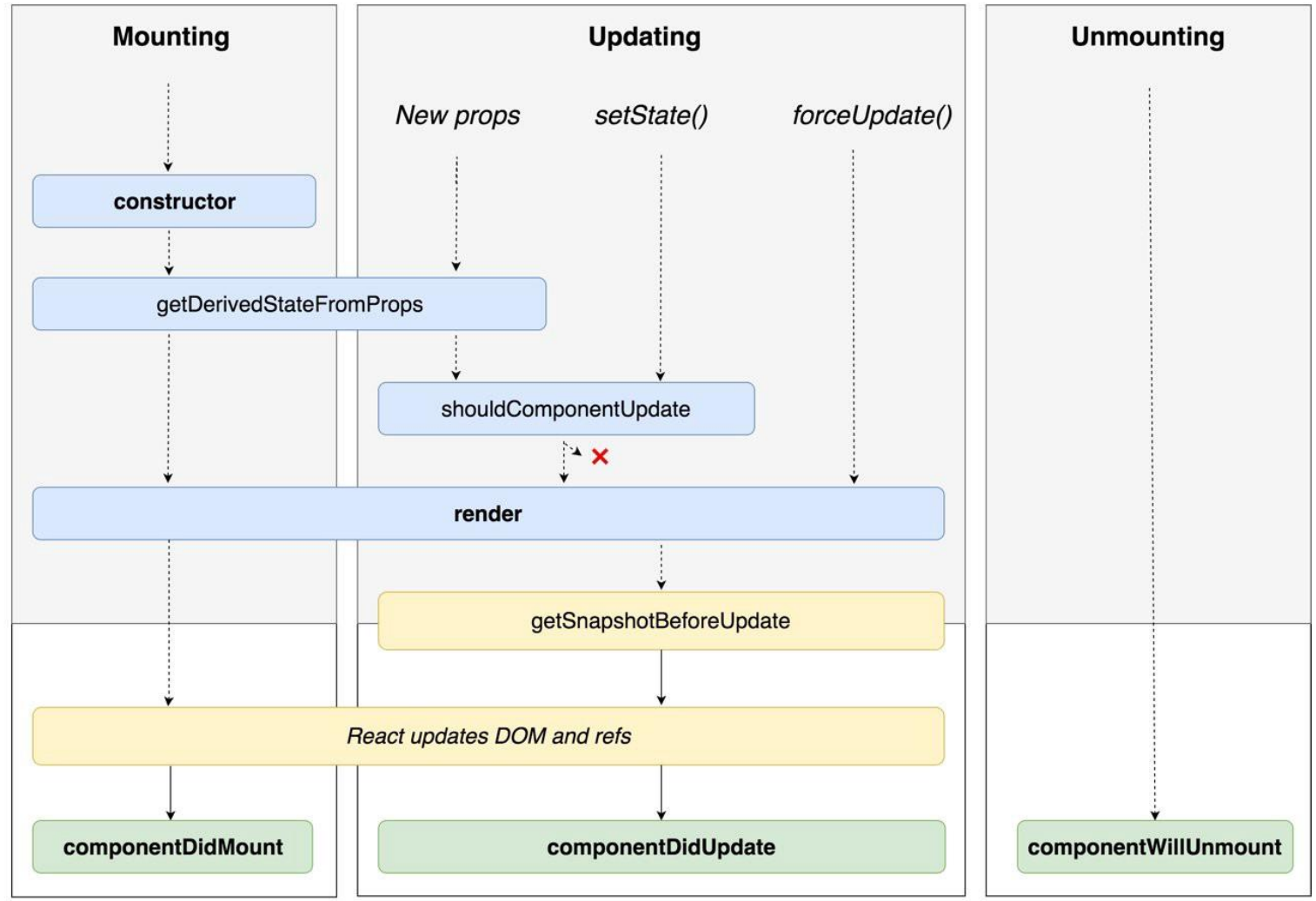
Pure and has no side effects. May be paused, aborted or restarted by React.

“Pre-Commit Phase”

Can read the DOM.

“Commit Phase”

Can work with DOM, run side effects, schedule updates.



componentDidMount

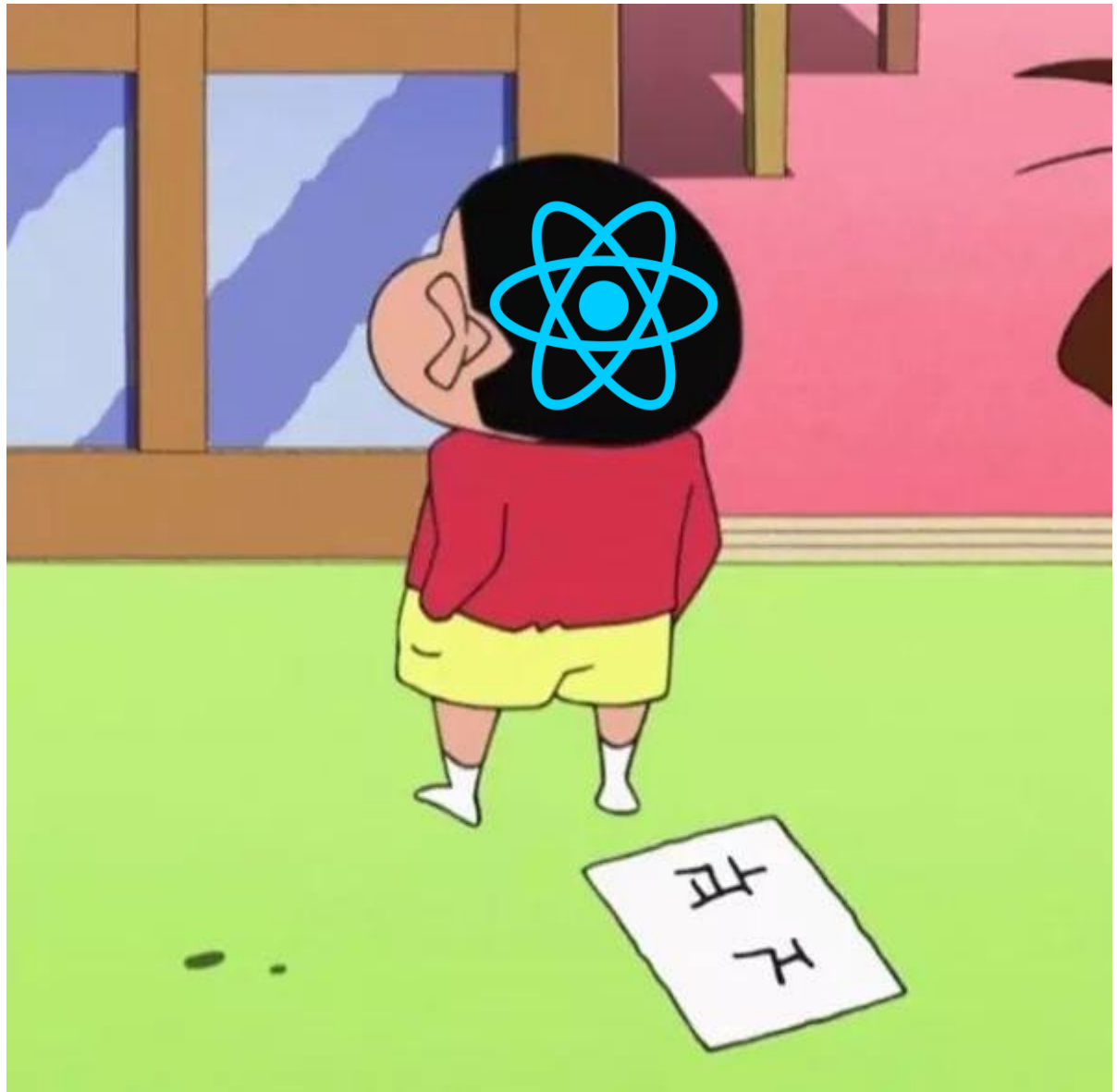
componentDidUpdate

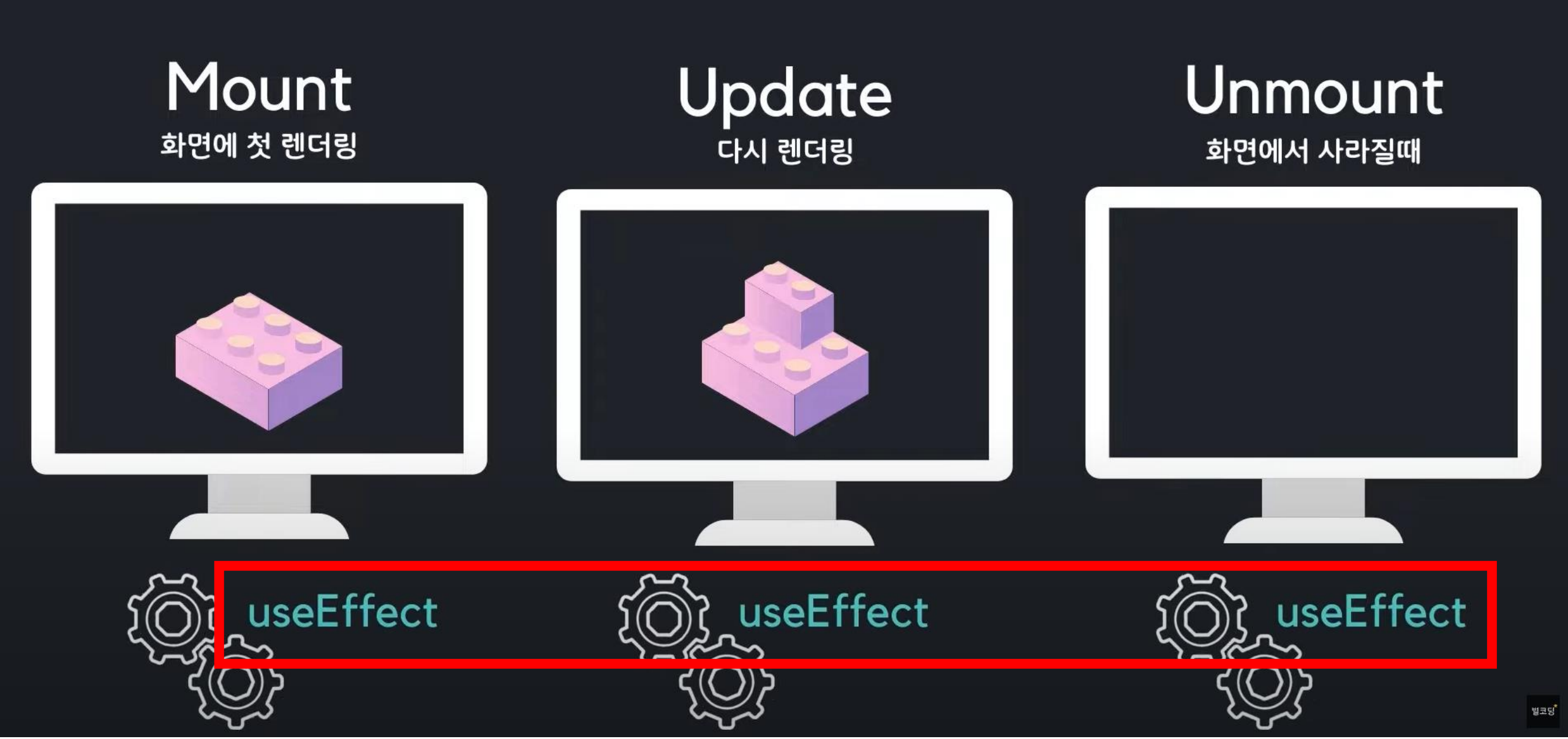
componentWillUnmount

```
class Cycle extends React.Component {
    componentWillMount(){
        alert("render전 가장 먼저 호출")
    }
    componentDidMount(){
        alert("ComponentDidMount호출 됨")
    }

    shouldComponentUpdate(nextProps, nextState){
        return this.props.name !== nextProps.name;
    }


    render(){
        alert("render 호출");
        return <h2>hello, {this.props.name}</h2>
    }
}
```

클래스형 컴포넌트의 Life Cycle

- 리액트는 위와 같은 기능을 통해서 컴포넌트가 처음 불려 왔을 때, 리렌더링 되었을 때, 컴포넌트가 사라질 때 마다 특정한 기능을 수행 할 수 있도록 처리를 했습니다!


과
거


Mount
화면에 첫 렌더링
Update
다시 렌더링
Unmount
화면에서 사라질때
useEffect
useEffect
useEffect

함수형 컴포넌트의 Life Cycle

- 하지만 우리 리액트에게 과거는 과거일 뿐 더 빠르고 편리한 미래만을 그립니다!
- 따라서, 클래스형 컴포넌트에서 함수형 컴포넌트로 넘어 가면서 기존의 Life CyIce 을 담당하던 기능을 하나의 HOOK에 추가 시켰습니다!
- 바로 UseEffect 입니다!

useEffect

별코딩

https://www.youtube.com/watch?v=kyodvzc5GHU

```
useEffect( ( ) => {

    // 작업...

} );
```

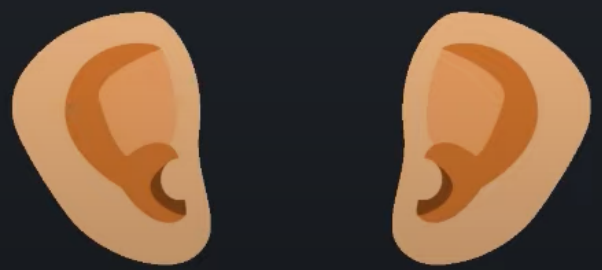
렌더링 될때 마다 실행

```
useEffect( ( ) => {
    // 작업...
}, [ value ] );
```

화면에 첫 렌더링 될때 실행

value 값이 바뀔때 실행

Clean Up - 정리



```
useEffect( ( ) => {

    // 구독 ...

    return ( ) => {

        // 구독 해지 ...
    }

}, [ ] );
```

그럼 코드로 한번 확인해 봅시다!

- 버튼을 클릭하면 count 가 +1 이 되어 숫자가 증가되는 컴포넌트를 만들어 봅시다!
- 그리고 useEffect 를 사용해서 state 값 변경에 따라 컴포넌트라 렌더링 될 때 마다 useEffect 가 작동하는지 알아 봅시다!

```jsx
import { useEffect, useState } from "react";

export default function TestUseEffect() {
    const [count, setCount] = useState(0);

    const onButtonClick = () => {
        console.log("👆 버튼 클릭");
        setCount(count + 1);
    }

    useEffect(() => {
        console.log("🎨 렌더링 할 때마다 실행되는 useEffect");
    })

    return (
        <>
            <h1>{count}</h1>
            <button onClick={onButtonClick}>+1 버튼</button>
        </>
    )
}
```

src/components/TestUseEffect.js

버튼 클릭
렌더링 할 때마다 실행되는 useEffect
버튼 클릭
렌더링 할 때마다 실행되는 useEffect
버튼 클릭
렌더링 할 때마다 실행되는 useEffect
버튼 클릭
렌더링 할 때마다 실행되는 useEffect
버튼 클릭
렌더링 할 때마다 실행되는 useEffect

그럼 코드로 한번 확인해 봅시다!

- 지금은 버튼이 클릭 되면 state 값의 변경이 일어나기 때문에 컴포넌트가 다시 렌더링이 되고, 그로 인해서 useEffect 내부의 함수가 실행 되고 있습니다!
- 그럼 여기에다가 input 태그를 하나 추가하고, input 태그의 입력 내용이 h1 태그에 출력 되도록 코드를 변경해 봅시다!

src/components/TestUseEffect.js

```jsx
import { useEffect, useRef, useState } from "react";

export default function TestUseEffect() {
    const [count, setCount] = useState(0);
    const [text, setText] = useState("입력 하세요!");
    const inputValue = useRef();

    const onButtonClick = () => {
        console.log("👆 버튼 클릭");
        setCount(count + 1);
    }

    const onInputChange = () => {
        console.log("💻 키 입력");
        setText(inputValue.current.value);
    }

    useEffect(() => {
        console.log("🎨 렌더링 할 때마다 실행되는 useEffect");
    })

    return (
        <>
            <h1>{count}</h1>
            <button onClick={onButtonClick}>+1 버튼</button>
            <br /><br /><br /><br />
            <input ref={inputValue} onChange={onInputChange}></input>
            <h1>{text}</h1>
        </>
    )
}
```

2

+1 버튼

qwe

qwe

```
버튼 클릭
렌더링 할 때마다 실행되는 useEffect
키 입력
렌더링 할 때마다 실행되는 useEffect
키 입력
렌더링 할 때마다 실행되는 useEffect
키 입력
렌더링 할 때마다 실행되는 useEffect
버튼 클릭
렌더링 할 때마다 실행되는 useEffect
```

useEffect

Dependency

```
useEffect( ( ) => {
    // 작업...
}, [ value ] );
```

화면에 첫 렌더링 될때 실행

value 값이 바뀔때 실행

useEffect 의 Dependency Array

- useEffect 는 두번째 인자로 Dependency Array 를 받습니다
- 해당 Array 에는 변수를 넣을 수가 있으며, 해당 변수가 변경 될 때에만 useEffect 내부의 함수가 실행 됩니다!
- + 빈 배열 [] 을 넣으면 최초 마운트 시에만 실행이 됩니다!
- 그럼 아래의 3가지 useEffect 코드를 추가한 뒤 테스트를 해봅시다!

```
useEffect(() => {
        console.log("🎨 렌더링 할 때마다 실행되는 useEffect");
    })

    useEffect(() => {
        console.log("👆 버튼 클릭 시에만 실행되는 useEffect");
    }, [count])

    useEffect(() => {
        console.log("⌨️ 인풋 입력 시에만 실행되는 useEffect");
    }, [text])
```

src/components/TestUseEffect.js

src/components/TestUseEffect.js

전체 코드

```jsx
import { useEffect, useRef, useState } from "react";

export default function TestUseEffect() {
    const [count, setCount] = useState(0);
    const [text, setText] = useState("입력 하세요!");
    const inputValue = useRef();

    const onButtonClick = () => {
        setCount(count + 1);
    }

    const onInputChange = () => {
        setText(inputValue.current.value);
    }

    useEffect(() => {
        console.log("🎨 렌더링 할 때마다 실행되는 useEffect");
    })

    useEffect(() => {
        console.log("👆 버튼 클릭 시에만 실행되는 useEffect");
    }, [count])

    useEffect(() => {
        console.log("⌨ 인풋 입력 시에만 실행되는 useEffect");
    }, [text])

    return (
        <>
            <h1>{count}</h1>
            <button onClick={onButtonClick}>+1 버튼</button>
            <br /><br /><br /><br />
            <input ref={inputValue} onChange={onInputChange}></input>
            <h1>{text}</h1>
        </>
    )
}
```

useEffect 의 Dependency Array

- 이제 버튼을 클릭하면 Dependency 로 count 를 전달한 useEffect 만 작동하고, 인풋에 값을 입력하면 Dependency 로 value 를 전달한 useEffect 만 작동하게 됩니다!
- 물론 Dependency Arr 를 전달하지 않은 경우는 렌더링 때마다 실행

```
🎨 렌더링 할 때마다 실행되는 useEffect
🖱️ 버튼 클릭 시에만 실행되는 useEffect
🎨 렌더링 할 때마다 실행되는 useEffect
⌨️ 인풋 입력 시에만 실행되는 useEffect
```

Dependency Array 를 전달하면?

- 두번째 인자 자체를 전달 하지 않으면, 매번 랜더링 마다 실행이 되지만 빈 배열을 두번째 인자로 전달 한다면?

```
useEffect(() => {
        console.log("ㅅ 최초 마운트 시에만 실행");
}, [])
```

- 요 친구는 변화를 감지할 값이 없으므로 최초 마운트 시에만 실행이 됩니다!

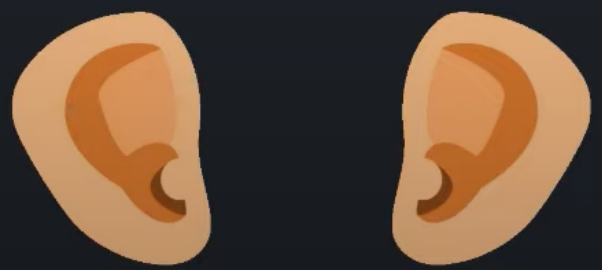
useEffect

Clean-Up

useEffect 정리하기

- 지금까지 useEffect 를 컴포넌트가 마운트 되는 순간과 리렌더링 되는 순간에 적용하여 사용하는 방법을 배웠습니다
- 그럼, 컴포넌트가 Unmount 되는 순간에는 어찌 처리하면 될까요?
- 클래스형에서는 componentWillUnmount 라는 메소드를 사용했지만 useEffect HOOK 에서는 useEffect 의 리턴에 함수를 부여하면 됩니다!

Clean Up - 정리



```
useEffect( ( ) => {

    // 구독 ...

    return ( ) => {

        // 구독 해지 ...
    }

}, [ ] );
```

코드로 보기!

- 간단한 Timer 컴포넌트를 만들어 봅시다!
- 버튼을 클릭하면 setInterval 함수를 통해 1초에 한번씩 console.log 를 찍는 컴포넌트 입니다!
- 그리고 해당 컴포넌트를 조건부 렌더링 처리하여 Mount 와 Unmount 를 시킬 수 있도록 만들어 봅시다!

```jsx
import { useEffect } from "react";

export default function Timer() {
    useEffect(() => {
        setInterval(() => {
            console.log(`타이머 실행중`)
        }, 1000);
    }, []);

    return (
        <>
            <h1>타이머가 실행 중입니다!</h1 >
        </>
    )
}
```

src/components/Timer.js

```jsx
import { useState } from 'react';

function App() {
  const [show, setShow] = useState(false);

  return (
    <div className="App">
      {show && <Timer />}
      <button onClick={() => setShow(!show)}>버튼</button>
    </div>
  );
}

export default App;
```

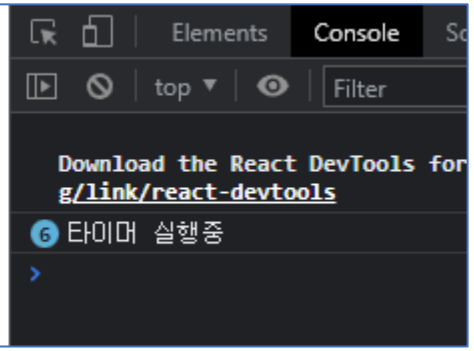
src/App.js

Timer 컴포넌트

- 버튼을 클릭하면 Timer 가 돌기 시작합니다!

타이머가 실행 중입니다!

버튼



Timer 컴포넌트

- 하지만 버튼을 다시 눌러서 Unmount 를 시켜 봅시다!

- 컴포넌트가 unmount 가 되어도 타이머는 계속 돌아가게 됩니다!

Clean-up

- 이런 상황을 방지하기 위해서 Unmount 가 되면 실행되는 Clean-up 을 이용, 타이머를 제거해 줍시다!
- 기존의 useEffect 코드에 return 으로 Clean-up 함수를 지정하여 줍시다

```jsx
import { useEffect } from "react";

export default function Timer() {
    useEffect(() => {
        const timer = setInterval(() => {
            console.log(`타이머 실행중`)
        }, 1000);

        return (() => {
            clearInterval(timer);
        })
    }, []);

    return (
        <>
            <h1>타이머가 실행 중입니다!</h1 >
        </>
    )
}
```

src/components/Timer.js

Clean-up

- 이제 Unmount 가 되는 상황에서는 return 에 인자로 전달한 함수가 실행이 되고 타이머가 정상 종료 됩니다!

실습, 컴포넌트 타이머!

- PracticeTimer 라는 컴포넌트를 만들어 줍시다!
- App.js 에는 '보이기' 라는 버튼이 하나 있습니다. 해당 버튼을 클릭하면 PracticeTimer 가 마운트 됩니다.
- 버튼을 클릭하면 PracticeTimer 컴포넌트가 마운트 되고 마운트가 된 시간을 초단위로 기록하는 타이머가 실행 됩니다.
- PracticeTimer 에도 '시간' 버튼이 존재하며, 해당 버튼을 누르면 지금까지 마운트가 된 시간을 출력해 줍니다!

실습, 컴포넌트 타이머!

- ‘보이기’ 버튼을 한번 더 클릭하면, PracticeTimer 가 Unmount 되고 타이머도 종료 되어야 합니다!
- 추가, ‘보이기‘ 버튼이 클릭 되면 버튼 이름을 ‘숨기기’로 변경 하기!
- 추가, 페이지가 처음 시작 되면 ‘보이기‘ 버튼에 포커스가 이동하도록 처리

보이기

0

시간 숨기기

Elemen
top
Download the R
g/link/react-d
1
2
3

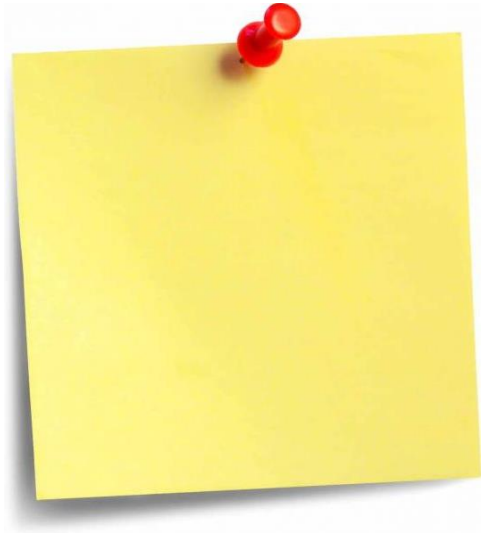
보이기

Elements Console
top Filter
Download the React DevTool
g/link/react-devtools
1
2
3
4
5
6
7
8
9
10
11


세상에서 제일 중요한 거거든
useState
useRef
useEffect

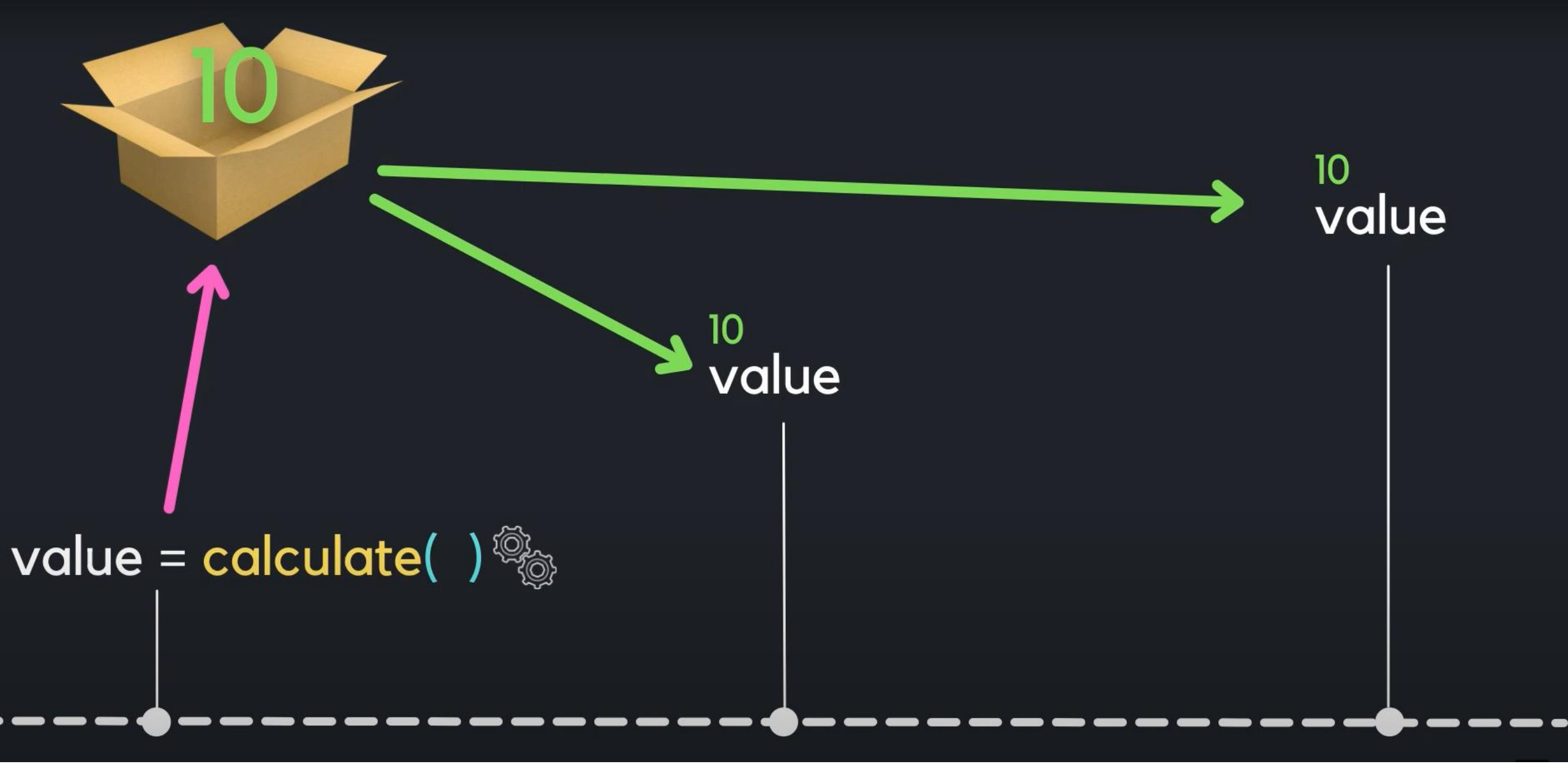

1
2
3
4
5
1
1등급에 가까울수록 에너지가 절약됩니다.
에너지소비효율등급

useMemo



useMemo?

- 저희는 언제 메모를 하나요!?
- 중요한 일을 까먹지 않기 위해서 합니다!
- 그런데 컴퓨터는 기억을 잘 하죠? 그럼 컴퓨터는 메모를 언제 쓸까요?
- 보통 오래 계산이 필요한 값을 반복적으로 사용하고 싶을 때, 해당 값을 저장해서 사용합니다!


10
10
value
10
value
value = calculate( )

함수형 컴포넌트

```
function Component ( ) {
    const value = calculate ( );
    return <div> { value } </div>
}
```

< Component />

렌더링

↓

Component 함수 호출

↓

모든 내부 변수 초기화

함수형 컴포넌트

```
function Component ( ) {

    const value = useMemo (

        ( ) => calculate ( ) , [ ]

    )


    return <div> { value } </div>

}
```

< Component />

렌더링

↓

Component 함수 호출,
Memoization

↓

렌더링

↓

Component 함수 호출,
Memoize 된 값을 제사용

useMemo

```
const value = useMemo(() => {
    return calculate();
}, [item]);
```

useMemo!

- 큰 계산이 걸리는 값을 미리 저장하여 컴포넌트가 다시 리렌더링 되었을 때 다시 계산 하는 것이 아니라, 메모리에 저장하여 바로 사용!
- 인자로는 uesEffect 와 동일하게, 첫 인자로는 Callback 함수를 두번째로는 의존성(dependency) 인자를 받습니다!
- 의존성은 useEffect 와 비슷하게 사용되는데요. 인자를 전달하면 해당 값이 업데이트 될 때 콜백 함수가 실행되고, 빈 배열을 전달하면 최초 마운트 시에만 실행이 됩니다!

useMemo,

꼭 필요할때만!

자 코드로 봅시다!

- TestUseMemo.js 라는 컴포넌트를 만들어 봅시다!
- For 문이 많이 돌아가는 함수로 계산이 오래 걸리는 함수를 만들어 봅시다!
- 그리고 바로 계산이 되는 심플한 함수도 만들어 봅시다!
- 이런 상태에서 심플한 함수만을 변경하였을 때, 컴포넌트가 리렌더링 되면서 오래 걸리는 함수까지 돌아가서 효율이 떨어지는 상황을 만들어 봅시다!

```jsx
import { useState } from "react";

export default function TestUseMemo() {
    const hardCalculate = (number) => {
        console.log("어려운 계산 시작!");
        let sum = 0;
        // 시간이 걸리는 계산
        for (let i = 1; i < 1000000001; i++) {
            sum = sum + 1;
        }
        return number + sum;
    }

    const [hardNum, setHardNum] = useState(1);
    const hardSum = hardCalculate(hardNum);

    const easyCalculate = (number) => {
        console.log("쉬운 계산 시작!");
        let sum = 1000000000;
        // 시간이 걸리는 계산
        return number + sum;
    }

    const [easyNum, setEasyNum] = useState(1);
    const easySum = easyCalculate(easyNum);
```

src/components/TestUseMemo.js

```
return (
    <>
        <h1> 시간이 오래 걸리는 계산</h1>
        <input type="number" value={hardNum} onChange={(e) => { setHardNum(parseInt(e.target.value)) }} />
        <span> + 100000000 = {hardSum}</span>
        <br />
        <h1> 시간이 안 걸리는 계산</h1>
        <input type="number" value={easyNum} onChange={(e) => { setEasyNum(parseInt(e.target.value)) }} />
        <span> + 100000000 = {easySum}</span>
    </>
  )
}
```

src/components/TestUseMemo.js

src/components/TestUseMemo.js

전체 코드

```jsx
import { useState } from "react";

export default function TestUseMemo() {
    const hardCalculate = (number) => {
        console.log("어려운 계산 시작!");
        let sum = 0;
        // 시간이 걸리는 계산
        for (let i = 1; i < 1000000001; i++) {
            sum = sum + 1;
        }
        return number + sum;
    }
    const [hardNum, setHardNum] = useState(1);
    const hardSum = hardCalculate(hardNum);

    // 시간이 안 걸리는 계산
    const easyCalculate = (number) => {
        console.log("쉬운 계산 시작!");
        let sum = 1000000000;
                return number + sum;
    }
    const [easyNum, setEasyNum] = useState(1);
    const easySum = easyCalculate(easyNum);

    return (
        <>
            <h1> 시간이 오래 걸리는 계산</h1>
            <input type="number" value={hardNum} onChange={(e) => { setHardNum(parseInt(e.target.value)) }} />
            <span> + 100000000 = {hardSum}</span>
            <br />
            <h1> 시간이 안 걸리는 계산</h1>
            <input type="number" value={easyNum} onChange={(e) => { setEasyNum(parseInt(e.target.value)) }} />
            <span> + 100000000 = {easySum}</span>
        </>
    )
}
```

시간이 오래 걸리는 계산

1 + 100000000 = 1000000001

시간이 안 걸리는 계산

1 + 100000000 = 1000000001

쉬운 계산도 시간이!?

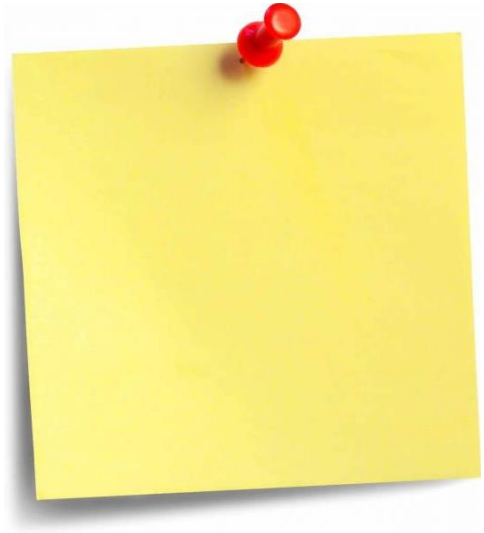
- 시간이 오래 걸리는 계산이야 그렇다고 쳐도 쉬운 계산에 의해 컴포넌트가 리렌더링 되어도 시간이 오래 걸리는 계산 때문에 마찬가지로 시간이 오래 걸립니다!
- 이럴 때 문제를 해결해 주는게 useMemo 가 됩니다!
- 먼저 어려운 계산의 결과값은 항상 100000 을 더하기 때문에 동일하고

useMemo!

- 먼저 어려운 계산의 결과값은 항상 100000 을 더하기 때문에 동일하기 때문에 useMemo 를 통해 저장해서 값을 사용하고
- useMemo 의 Dependency 를 이용해서 hardNum 이 변경이 될 때에만 컴포넌트를 다시 그려 봅시다!

```jsx
export default function TestUseMemo() {
    const hardCalculate = (number) => {
        console.log("어려운 계산 시작!");
        let sum = 0;
        // 시간이 걸리는 계산
        for (let i = 1; i < 1000000001; i++) {
            sum = sum + 1;
        }
        return number + sum;
    }

    const [hardNum, setHardNum] = useState(1);
    const hardSum = useMemo(() => {
        return hardCalculate(hardNum);
    }, [hardNum])

    const easyCalculate = (number) => {
        console.log("쉬운 계산 시작!");
        let sum = 1000000000;
        // 시간이 걸리는 계산
        return number + sum;
    }

    const [easyNum, setEasyNum] = useState(1);
    const easySum = easyCalculate(easyNum);
```

src/components/TestUseMemo.js



useMemo

실전 활용!

useMemo 실전 활용

- 사실 지금까지 쓴 useMemo 는 useEffect 와 크게 다를 바가 없습니다!
- 물론, 메모리 적으로 약간 이득은 봤습니다.
- 다만 실상 저희가 이용한 건 Dependency 배열을 사용하는게 거의 전부 였을 뿐입니다!

useMemo 실전 활용

- 그럼 실제적으로 useMemo는 어디에서 사용을 할까요?
- 숫자를 올릴 수 있는 인풋과, 현재 상태에 따라 한국 외국을 구분하는 컴포넌트를 만들어 봅시다
- 그리고 useEffect 의 의존성 배열로 현재 상태를 줘 봅시다!

```jsx
import { useEffect, useState } from "react";

export default function UsingUseMemo() {
    const [number, setNumber] = useState(0);
    const [isKorea, setIsKorea] = useState(true);

    const location = isKorea ? "한국" : "외국";

    useEffect(() => {
        console.log("✨ useEffect 호출!");
    }, [location])

    return (
        <>
            <h1>숫자 증감</h1>
            <input type="number" value={number} onChange={(e) =>
setNumber(parseInt(e.target.value))} />
            <br />
            <h1>Where are you?</h1>
            <h2>위치: {location}</h2>
            <button onClick={() => setIsKorea(!isKorea)}>나라 변경</button>
        </>
    )
}
```

src/components/UsingUseMemo.js

useMemo 실전 활용

- 현재 상태는 나라 변경 버튼으로 location 값이 변경 되어야만 useEffect 에 전달 된 의존성 배열에 의해 useEffect 가 호출이 되고 있습니다!
- 그럼, 여기서 location 을 일반 변수에서 객체로 변경해 봅시다!

```
const location = {
        where: isKorea ? "한국" : "외국",
    }

… 다른 코드들

 <h2>위치: {location.where}</h2>
```

src/components/UsingUseMemo.js

useMemo 실전 활용

- 이 상태에서는 이전과는 달리 숫자를 변경해도 useEffect 가 호출이 됩니다!
- 그리고 useEffect 에 노란색 줄이 가 있네요?

```
(property) where: string

The 'location' object makes the dependencies of useEffect Hook (at
line 13) change on every render. To fix this, wrap the
initialization of 'location' in its own useMemo()
Hook. eslint(react-hooks/exhaustive-deps)

문제 보기   빠른 수정... (Ctrl+.)

        where: isKorea ? "한국" : "외국",
    }
```

Quiz. 왜 그럴까요?

- 리액트 Eslint 가 미리 경고를 해주네요. 오브젝트를 의존성 배열로 전달하면 무조건 useEffect 가 불려오게 되니 useMemo 사용을 권하네요!
- 왜 그럴까요?

원시 (Primitive) 타입

String
Number
Boolean
Null
Undefined
BigInt
Symbol

객체 (Object) 타입

원시 타입을 제외한 모든 것

Object
Array
...

원시 (Primitive) 타입

```
const location = "korea"
```

location

"korea"

객체 (Object) 타입

```
const location = {
    country: "korea"

}
```

location

#12345

#12345

{ country: "korea" }

원시 (Primitive) 타입

```
const locationOne = "korea"
const locationTwo = "korea"



locationOne === locationTwo
> true
```

객체 (Object) 타입

```
const locationOne = {
    country: "korea"
}

const locationTwo = {
    country: "korea"
}


locationOne === locationTwo
> false
```

그리하여

- State 변경으로 인해 컴포넌트가 리렌더링이 되면 location 이라는 객체는 다시 할당을 받게 됩니다
- 따라서, 메모리 주소가 변하게 되고 이전에 의존성 배열로 전달한 location 객체와는 다른 주소를 가지게 됩니다!
- 그렇기 때문에 의존성 배열의 location 과 새로 할당 받은 location 의 비교 값은 false 가 되고 이로 인해서 location 값이 변했다고 useEffect는 판단을 합니다! → useEffect 가 호출이 되게 됩니다!

useMemo

- 하지만 useMemo 는 리턴한 값을 저장해 두고 다시 불러오기 때문에 컴포넌트가 다시 랜더링이 되어도 한번 저장한 값의 위치는 고정이 되게 됩니다!
- 따라서, 이전의 useEffect 같은 문제가 발생하지 않습니다!
- 그럼 location 객체를 useMemo 를 통해 저장해서 위와 같은 문제를 막아봅시다!

src/components/UsingUseMemo.js

```jsx
import { useEffect, useMemo, useState } from "react";

export default function UsingUseMemo() {
    const [number, setNumber] = useState(0);
    const [isKorea, setIsKorea] = useState(true);

    const location = useMemo(() => {
        return {
            where: isKorea ? "한국" : "외국",
        }
    }, [isKorea]);

    useEffect(() => {
        console.log("✨ useEffect 호출!");
    }, [location])

    return (
        <>
            <h1>숫자 증감</h1>
            <input type="number" value={number} onChange={(e) => setNumber(parseInt(e.target.value))} />
            <br />
            <h1>Where are you?</h1>
            <h2>위치: {location.where}</h2>
            <button onClick={() => setIsKorea(!isKorea)}>나라 변경</button>
        </>
    )
}
```


수고하셨습니다!