Seoul
Software
ACademy

with

컴포넌트로

코드를 줄이자~!

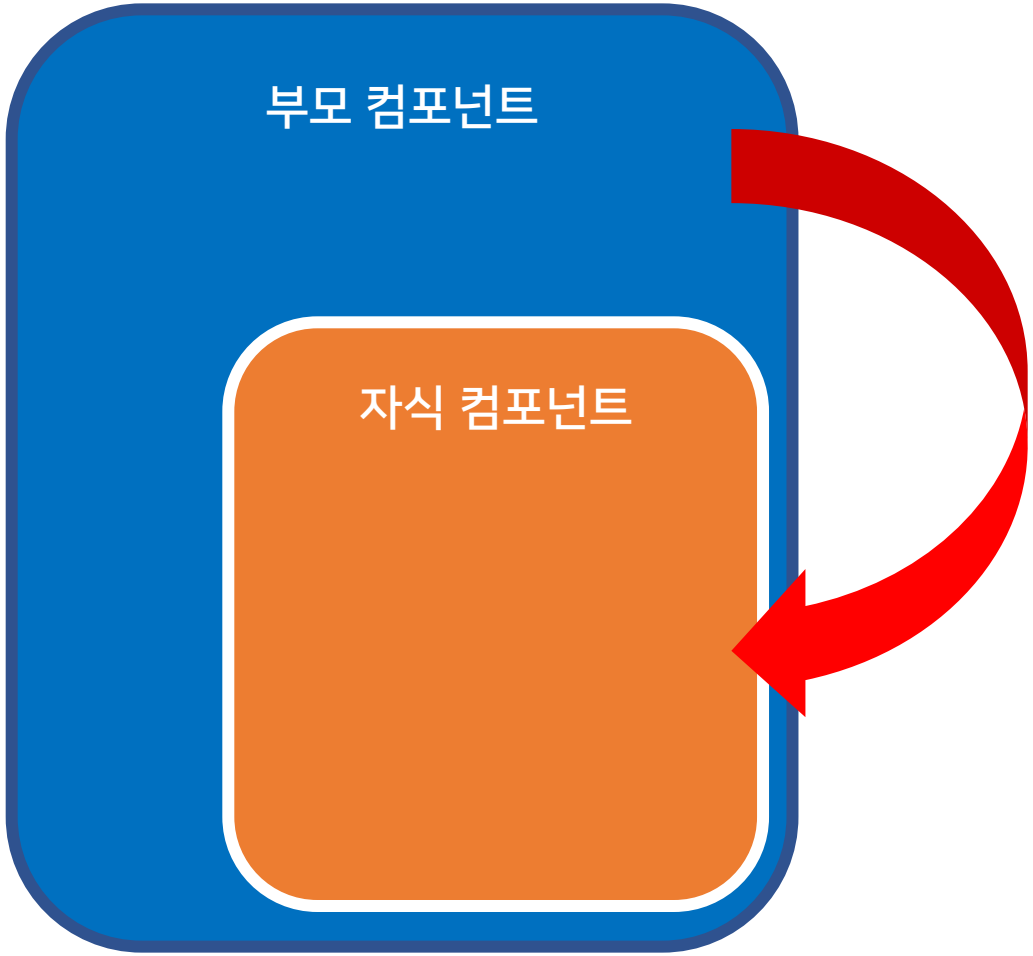
```jsx
function Modal() {
  return (
    <div className="modal">
      <h2>오늘 해야할 일 2개</h2>
      <h2>오늘 완료한 일 0개</h2>
    </div>
  );
}
export default Modal;
```

```jsx
import Modal from "./Modal";

function List() {
  return (
    <div>
      <h1>오늘 해야할일</h1>
      <Modal />
      <hr />
      <h2>리액트 공부하기</h2>
      <p>state 사용법 익히기</p>
      <hr />
      <h2>코테 문제 풀기</h2>
      <p>Lv 0 정복 가즈아</p>
      <hr />
      <Modal />
    </div>
  );
}
export default List;
```

Props

Props?

- 부모 컴포넌트에서 자식 컴포넌트에 원하는 데이터를 보내는 방법입니다!



- props 라는 이름의 객체로 전달

간단한 MainHeader 라는 컴포넌트 만들기

- MainHeader 의 매개변수 전달 부분에 {} 를 추가하고 부모로 부터 받아올 props 명을 적어 주시면 됩니다!

```
import React from "react";

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

export default MainHeader;
```

```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!

다양한 데이터 받아오기!

- Props 는 다양한 데이터를 한꺼번에 받아 올 수 있습니다!
- 그리고 다양한 데이터는 props 라는 객체 하나로 받아서 사용이 가능합니다!

```
function App() {
  return (
    <div className="App">
      <MainHeader text="Go naver" href="https://naver.com" userID="tetz!" />
    </div>
  );
}

export default App;
```

src/App.js

```jsx
function MainHeader(props) {
    return (
        <div>
            <h1>{props.userID} 님 반갑습니다.</h1>
            <a href={props.href}>{props.text}</a>
        </div>

    )
}
```

src/components/MainHeader.js

- 물론 구조 분해 할당 문법도 사용이 가능합니다!

리액트에서

배열 활용하기!

배열 데이터를 리액트로 그리는 방법!

- React 에서는 배열 데이터를 그려 줄 때 배열 함수인 map 을 사용합니다!

```
return (
  <div>
    <h1>오늘 해야할일</h1>
    <hr />
    {dataArr.map((el) => {
      return <ListChild title={el.title} detail={el.detail} />;
    })}
  </div>
);
```

```jsx
import ListChild from "./ListChild";
function List() {
  const dataArr = [
    {
      title: "리액트 공부하기",
      detail: "state 사용법 익히기",
    },
    {
      title: "코테 문제 풀기",
      detail: "Lv 0 정복 가즈아",
    },
  ];
  return (
    <div>
      <h1>오늘 해야할일</h1>
      <hr />
      {dataArr.map((el, index) => {
        return <ListChild title={el.title} detail={el.detail} key={index} />;
      })}
    </div>
  );
}
export default List;
```

구별이 가능한 unique 한 key 값을

props 로 부여하기!

단, index 는 최후의 수단으로 사용!

Props

활용하기

배열을 전달하고 props 로 받아서 처리!

- Props 로는 배열 같은 다양한 자료형도 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 객체를 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

안전하게

사용하기!

IF문 사용하기!

- If 문을 사용해서 props 가 들어 왔는지 확인 후, 처리를 해줍시다
- 단, if 문은 JSX 문법 내부(return 내부)에서는 사용할 수 없습니다! → 그래서 3항 연산자를 사용했죠!
- 대신 return 밖에서 사용을 하고 상황에 따라 다른 return 을 주는 방식으로는 가능합니다!

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

데이터가 없습니다.

Prototype Chaining 에 ? 사용하기

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

```
›
```


지금 시작합니다

useRef

useRef? 이 친구는 또 뭐죠?

- 무언가 컴포넌트가 리렌더링 되어도 값을 유지하고 싶을 때가 있습니다!
- 어? 그럼 state 쓰면 되는거 아닌가요?
- 그런데 값이 변경 되어도 컴포넌트 리렌더링을 하고 싶지는 않아요!
- 변수를 쓰면? → 리렌더링 시 변수 값은 초기화가 됩니다!
- 이럴 때 쓰는 것이 useRef 입니다!

useRef? 이 친구는 또 뭐죠?

- 자 Input 태그를 사용해서 무언가를 입력 받는 상황을 그려 봅시다!
- 만약 이럴 때, useState 를 사용한다면?
- 인풋 태그 자체가 계속 리렌더링이 되어서 문제 + 낭비가 생길 겁니다!
- 이럴 때 쓰라고 있는 것이 useRef 되겠습니다!

useRef

- useRef 를 사용하면 참조하고자 하는 DOM 요소에 ref 속성을 주고 해당 태그의 변화를 감지하거나 DOM 요소를 컨트롤 할 수 있습니다
- 보통은 컴포넌트에 존재하는 인풋 태그의 값을 받거나, JS에서 DOM 요소를 관리하던 역할을 해줍니다!

JS 방식으로 구현하기

- Components 폴더에 TestRef.js 컴포넌트를 작성해 주세요!
- 인풋에 입력 된 값이, 상단에 있는 h1 태그의 컨텐츠가 되도록 구성하여 봅시다!
- 일단은, 기존의 JS 처럼 onChange 속성을 사용해서 이벤트 객체를 이용하여 구현하여 봅시다!

```jsx
import TestRef from './components/TestRef';

function App() {
  return (
    <div className="App">
      <TestRef />
    </div>
  );
}

export default App;
```

src/App.js

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

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

    const onChangeText = (e) => {
        const inputText = e.target.value;
        setText(inputText);
    }

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

dasdadadada

dasdadadada

src/components/TestRef.js

useRef 로 구현하기

- useRef 를 사용하여 변화를 감지하거나 DOM 요소를 컨트롤 하고 싶은 태그에 ref 속성을 부여하기!

```
const ref = useRef(value)
```

```
<input ref={ ref } />
```

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

export default function TestUseRef() {
    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/TestUseRef.js

useRef 값 확인하기

- useRef 로 설정한 값을 console.log 에 찍어 봅시다!

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

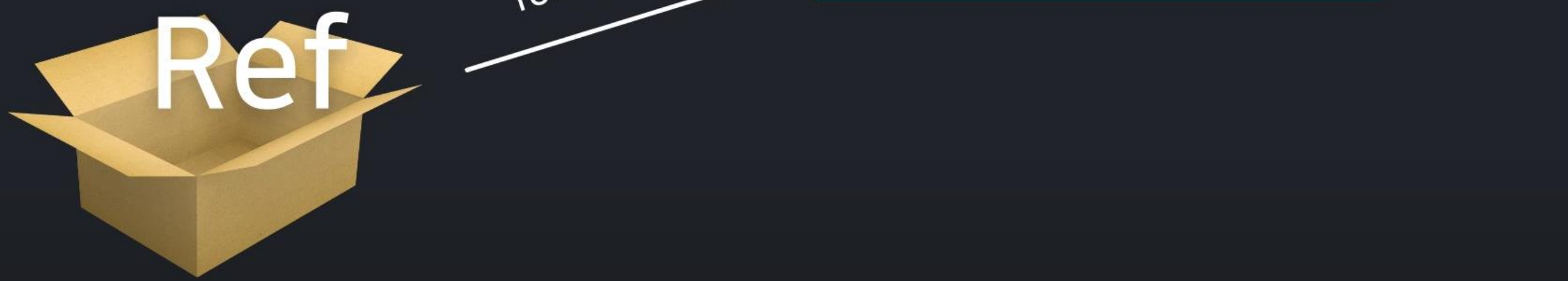
```
▼ {current: input} i
  ▼ current: input
      value: "dsadad"
    ▶ __reactEvents$3ew9d0gbjvx: Set(1) {'invalid__bubble'}
    ▶ __reactFiber$3ew9d0gbjvx: FiberNode {tag: 5, key: null, el
    ▶ __reactProps$3ew9d0gbjvx: {onChange: f}
    ▶ _valueTracker: {getValue: f, setValue: f, stopTracking: f}
    ▶ _wrapperState: {initialChecked: undefined, initialValue: '
```

useRef 로

포커스 이동 시키기!

useRef 로 포커스 이동!

- useRef 가 자주 사용되는 포커스 사용 방법을 알아 봅시다!
- 컴포넌트 폴더에 ChangeFocus.js 컴포넌트를 만들어 봅시다!
- 2개의 인풋 태그를 만들고 해당 인풋에 ref 속성을 부여!
- useRef 로 각각 인풋의 속성 값을 변수에 담고 해당 변수를 통해 input 태그에 포커스를 부여해 봅시다!
- 해당 값에 대한 접근은 current 객체를 통해 해야합니다!


Ref
focus ()
hello123
로그인

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

export default function ChangeFocus() {
    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

useRef 로

DOM 컨트롤

```
const ref = useRef(value)
```

```
<input ref={ ref } />
```

useRef 를 사용하여 DOM 컨틀롤!

- useRef 로 선언한 변수를 DOM 요소에 ref 속성으로 부여하면 해당 요소에 접근할 수 있습니다
- 마치 이전 VanilaJS 의 querySelector 또는 getElementById 같은 역할을 손쉽게 구현할 수 있죠! → 2개를 대체해서 사용이 가능합니다!
- 그럼 ref 로 접근한 DOM 요소를 컨트롤 해봅시다!

useRef 를 사용하여 DOM 컨틀롤!

- H1 태그 2개를 사용하고 useRef 로 DOM 요소에 접근하여 해당 DOM 요소의 스타일을 변경해 보겠습니다!
- 인풋 태그를 초기화도 해보죠!

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

export default function RefDOM() {
  const orangeEl = useRef();
  const skyblueEl = useRef();
  const inputEl = useRef();

  const adjustCSS = () => {
    orangeEl.current.style.backgroundColor = "orange";
    skyblueEl.current.style.backgroundColor = "skyblue";
  };

  const clearInput = () => {
    inputEl.current.value = "";
  };

  return (
    <div>
      <h1 ref={orangeEl}>Orange</h1>
      <h1 ref={skyblueEl}>Skyblue</h1>
      <input type="text" ref={inputEl} />
      <br />
      <button onClick={adjustCSS}>CSS 적용</button>
      <button onClick={clearInput}>인풋 초기화</button>
    </div>
  );
}
```

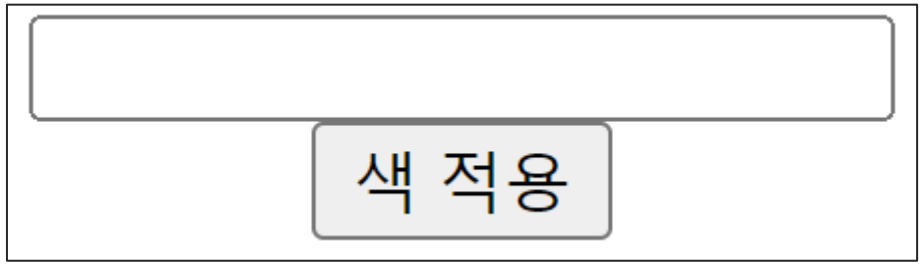
실습, useRef 활용하기!

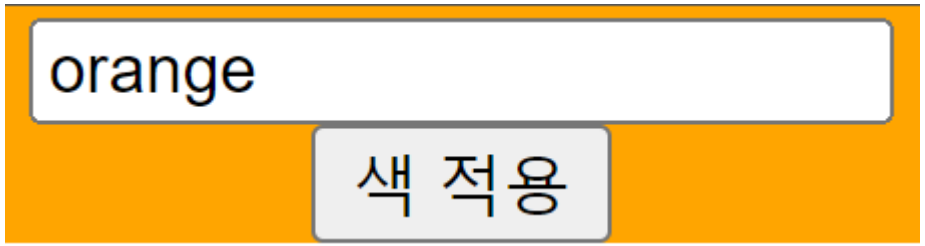
- DIV 요소의 배경색을 직접 입력 받아서 변경하는 ColorInput.js 컴포넌트를 만들어 봅시다~!

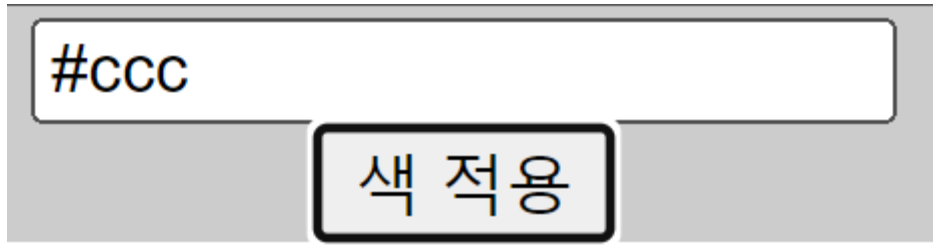
```
import React, { useRef } from 'react'

export default function ColorInput() {
  return (
    <div>
      <input />
      <br />
      <button>색 적용</button>
    </div>
  )
}
```

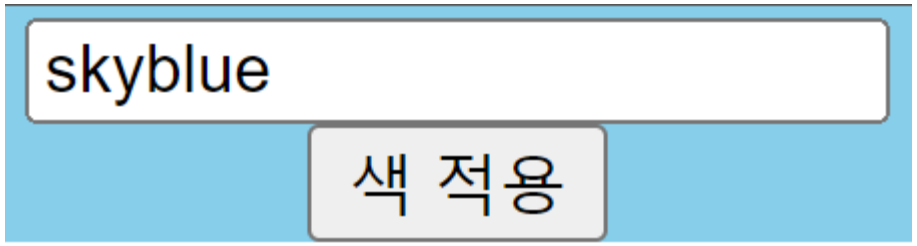
실습, useRef 활용하기!

- 위의 코드를 변경해서 Input 창에 색을 입력하고 색 변경 버튼을 누르면 컴포넌트의 배경색이 변경되는 컴포넌트를 완성해 주세요~!
- Input 값을 받는 것과 DOM 컨트롤은 useRef 를 활용해서 구현해야 합니다!





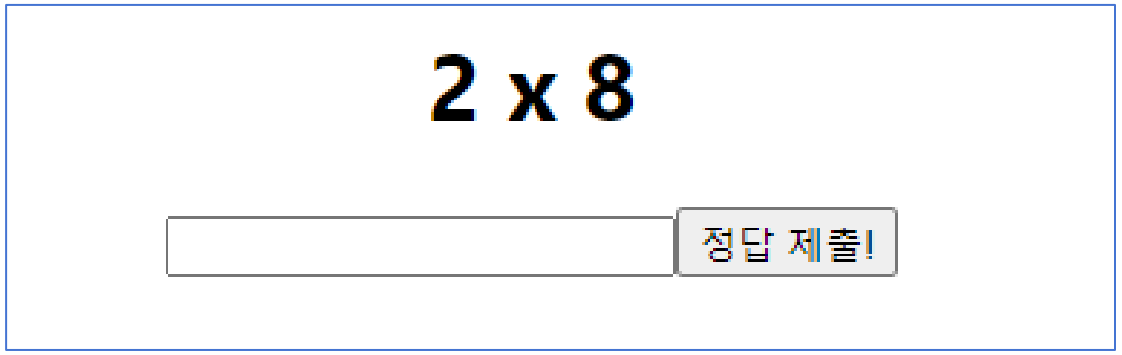





STARBUCKS
COFFEE
STARBUCKS
COFFEE

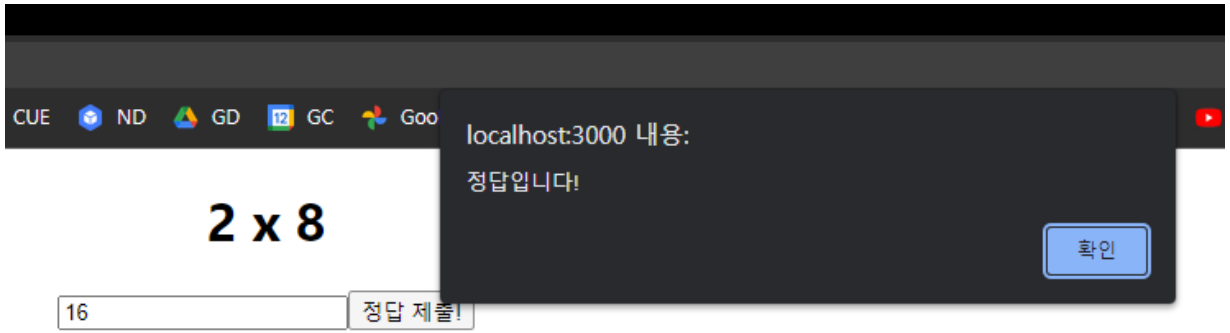
실습, 퀴즈 프로그램 만들기!

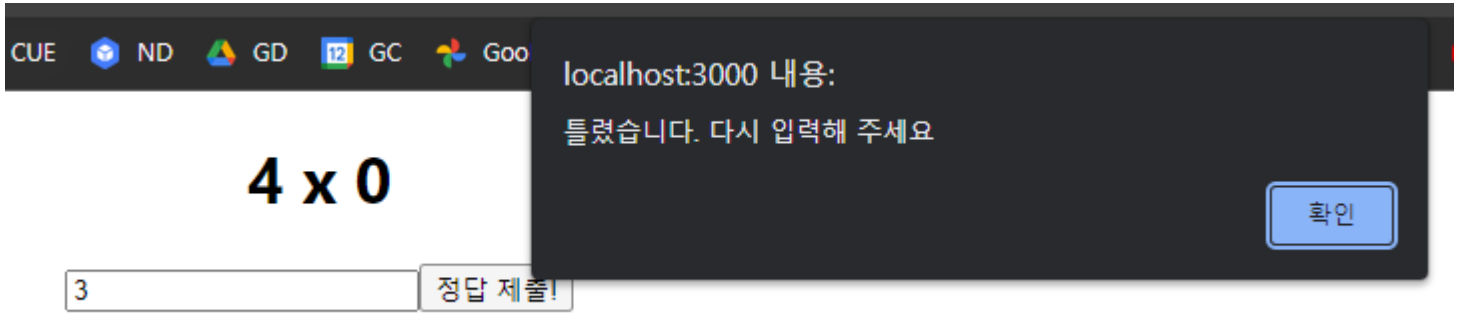
- 숫자 퀴즈 프로그램을 만들어 봅시다!
- 컴포넌트가 랜더링이 되면 0 ~ 9 사이의 수를 랜덤으로 2개 생성합니다.
- 그리고 랜덤으로 더하기, 빼기, 곱하기 중에서 어떤 연산을 할지 정하여 퀴즈를 출제합니다. (나눗셈은 소수가 나오므로 X)



실습, 퀴즈 프로그램 만들기!

- useRef 를 활용하여 input 태그의 값을 입력 받아서 퀴즈의 정답이 맞을 경우, 틀릴 경우 각각 alert 을 띄워 줍니다





실습, 퀴즈 프로그램 만들기!

- 정답을 입력한 경우 input 태그의 내용을 비워주고, input 태그로 포커스를 이동 시킨 뒤 컴포넌트를 리랜더링 하여 새로운 퀴즈를 출제합니다
- 오답을 입력한 경우 input 태그의 내용을 비워주고, input 태그로 포커스를 이동 시켜 줍니다
- 나누기는 소수점 계산이 필요하여 제외 합니다
- 답은 음수가 될 수 있습니다!

useState

useRef

Variable

3가지 타입에 대해서 정리해 봅시다!

- 지금까지 배운 useState 의 state 그리고 useRef 의 ref 와 리액트 내부의 변수가 렌더링에 따라 어떤 식으로 변화하는지 알아 봅시다!
- 그리고 state 에 변화에 따른 컴포넌트 리렌더링의 개념도 다시 한번 잡아 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!</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

```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;
        con sole.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;
```

State: 4

Ref: 14

Variable: 0

State UP! | Ref UP! | Variable UP! | 렌더링!

src/components/Comparing.js

React.Fragment

React.Fragment?

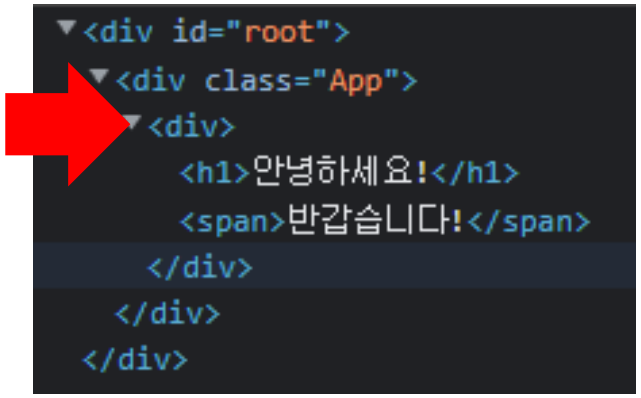
- 자, 정말 간단한 컴포넌트를 만들어 볼게요!

```
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
"./TableColumn";

function Table() {
  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 Table;
```

```jsx
import React from "react";

export default function TableColumn() {
  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
>
```

```jsx
import React from 'react’;

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

src/components/TableColumn.js

```
                                    react-dom.development.js:29840
  Download the React DevTools for a better development experience: http
  s://reactjs.org/link/react-devtools
>
```

조건부 렌더링!

컴포넌트를 상황에 따라 켜고 끄기!

- 상황에 따라서 컴포넌트를 보여줄지 여부를 정해야 할 때가 있습니다!
- 그럴 때 사용하는 것이 조건부 렌더링 입니다!
- 이전 JS, HTML 에서는 Display 속성을 none 으로 해서 처리하곤 했습니다
- 리액트는 JSX 문법을 사용 하므로, if 문 또는 3항 연산자, 논리 연산자와 HTML 태그를 같이 쓰면 되기 때문에 상당히 쉽습니다!
- 그럼 일단 한번 보시죠!

조건부 렌더링을 위한 컴포넌트 만들기

- Components 폴더에 Item.jsx 파일 만들기

```
function Item() {
    return (
        <h1>보이나요?</h1>
    )
}
export default Item;
```

src/components/Item.jsx

ConditionalRender 에서 조건부 렌더링

- ConditionalRender.jsx 에서 useState 를 활용해서 condition 에 따른 조건부 렌더링을 처리해 봅시다!
- Condition 이 감추기이면 Item 를 보여주고, 보이기이면 감춰 봅시다!
- 이럴 땐, 보통 && 연산자를 사용합니다!

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

function ConditionalRender() {
  const [condition, setCondition] = useState('보이기');

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

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

export default ConditionalRender;
```

src/components/ConditionalRender.jsx

변수로 처리해서 코드 정리!

- 조건부 렌더링 자체를 변수에 넣어서 처리해 봅시다!
- 이렇게 하면 코드가 깔끔해 지는 효과가 있습니다!
- 재사용에 유리합니다!
- 이런 것이 가능한 이유는 바로! → JSX 문법 덕분이죠!

```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");
  }

  const conditionRender = condition === "감추기" && <ConditionalRender />;

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

export default App;
```

src/App.js

실습, 조건부 렌더링 처리!

- PracticeOne, PracticeTwo 컴포넌트를 만들어 주세요!
- props 로 데이터를 받아서 h1 태그로 출력하는 간단한 구조를 가집니다!
- ExConditional.js 에서 버튼을 클릭하면 PracticeOne 과 PracticeTwo 가 번갈아서 렌더링 되는 조건부 렌더링 처리를 해주세요!
- 버튼의 컨텐츠도 렌더링 되는 캄포넌트 번호가 나오게 해주시면 됩니다!

1번 컴포넌트

1번

2번 컴포넌트

2번

Life Cycle

Mount

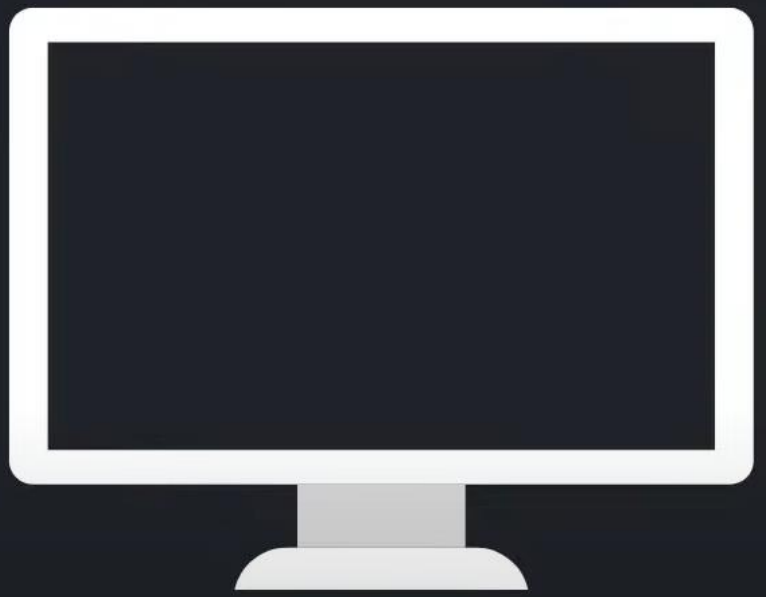
화면에 첫 렌더링

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”

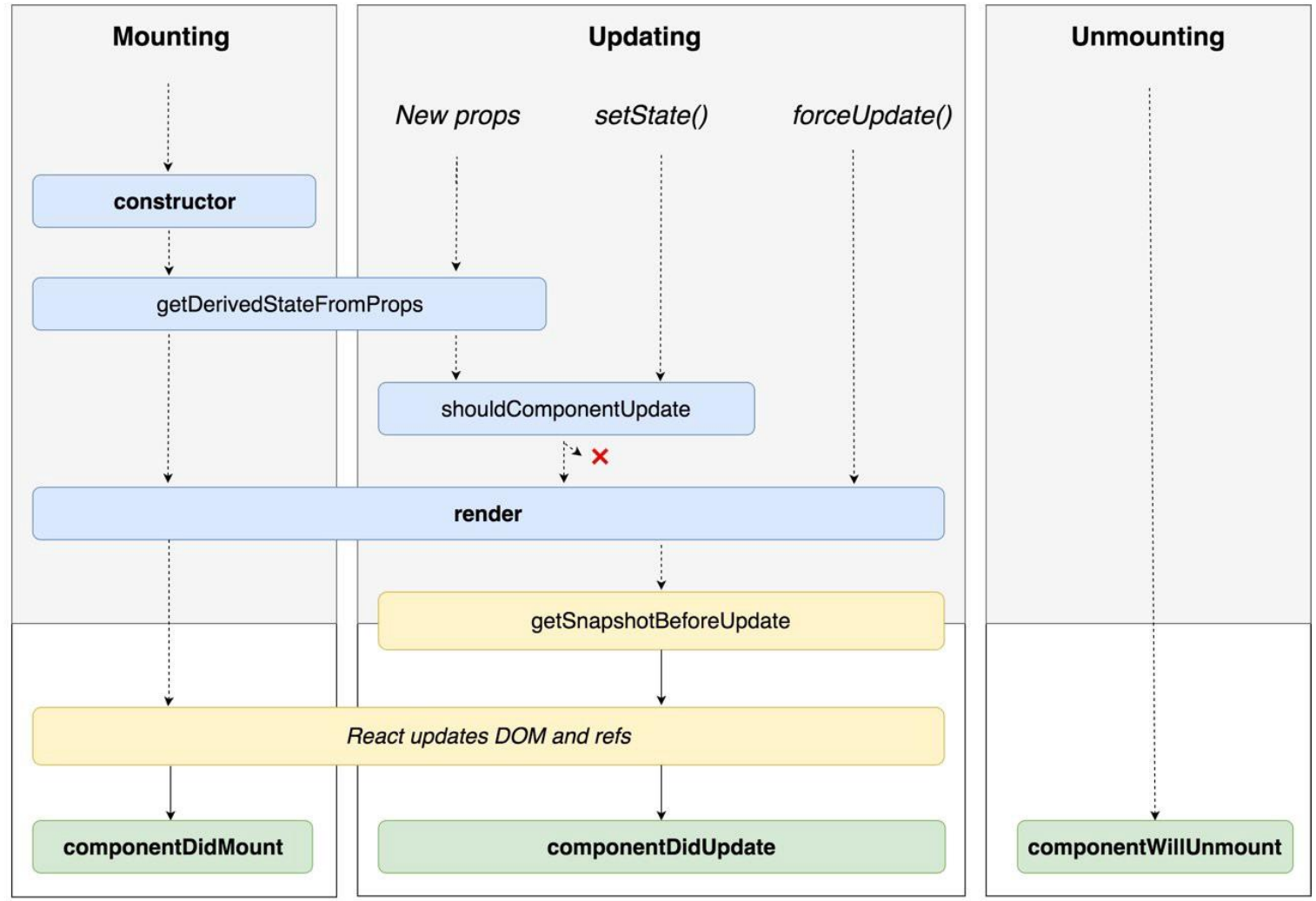
Pure and has no side effects. May be paused, aborted or restarted by React.

“Pre-Commit Phase”

Can read the DOM.

“Commit Phase”

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



componentDidMount

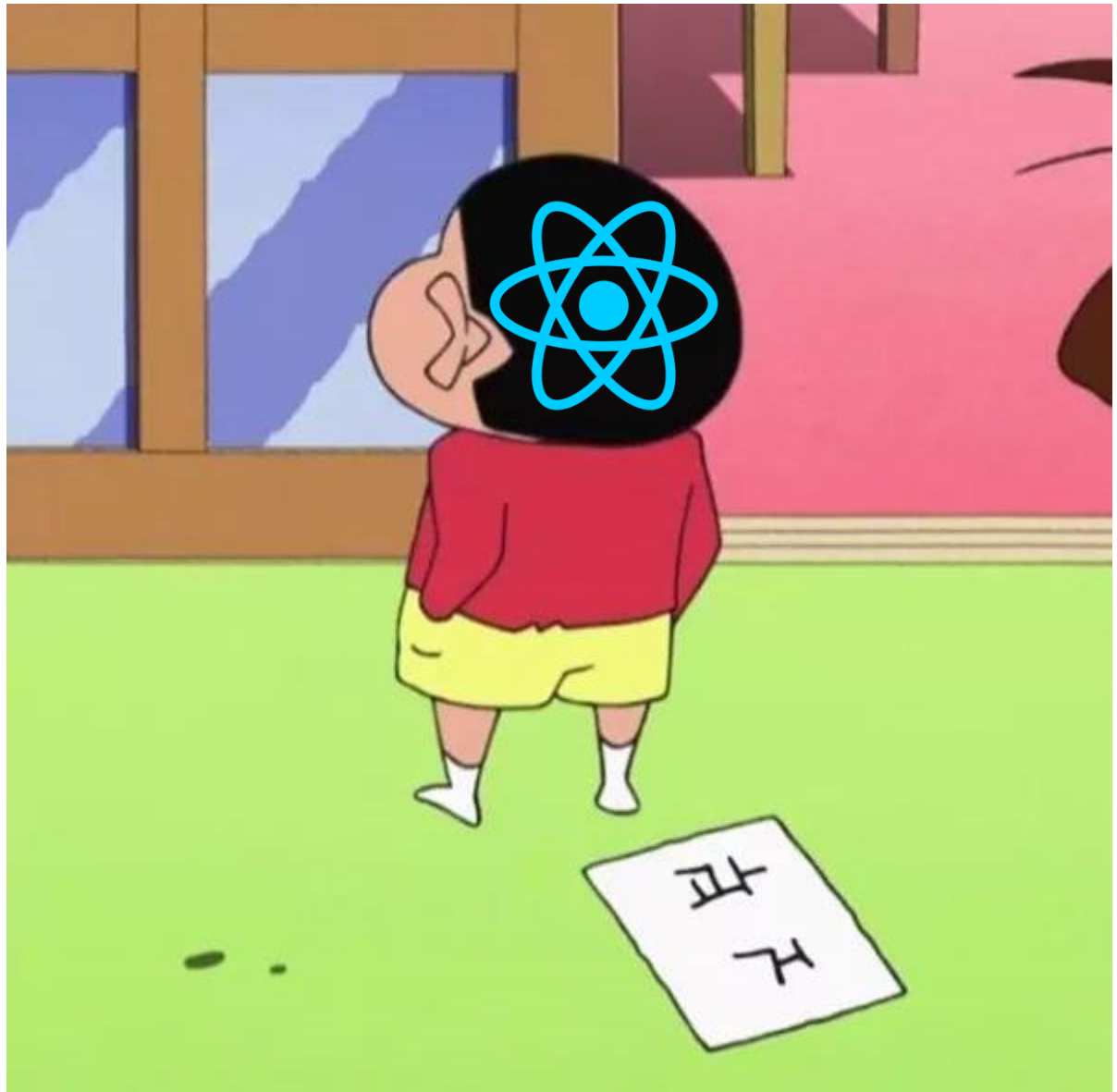
componentDidUpdate

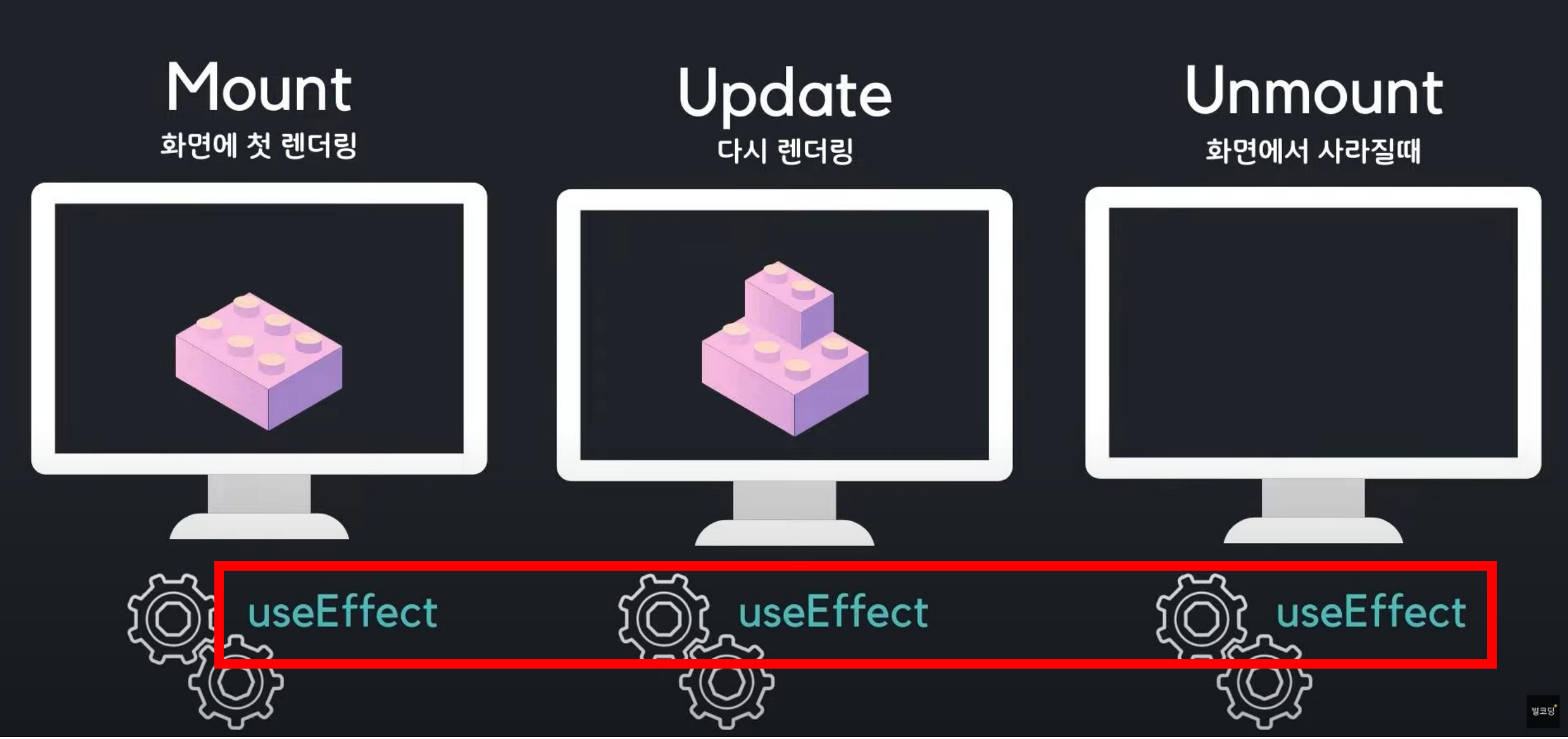
componentWillUnmount

```
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 Cylce 을 담당하던 기능을 하나의 HOOKS 에 추가 시켰습니다!
- 바로 UseEffect 입니다!

useEffect

별코딩⭐

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

```
useEffect( ( ) => {

    // 작업...

} );
```

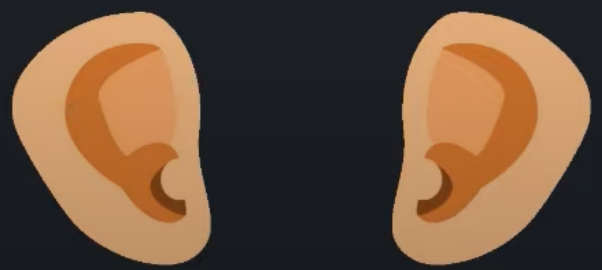
렌더링 될때 마다 실행

```
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!

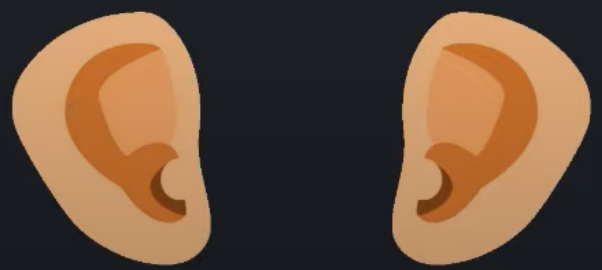
useEffect

Clean-Up

컴포넌트unmount 에 실행되는 useEffect

- 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;
```

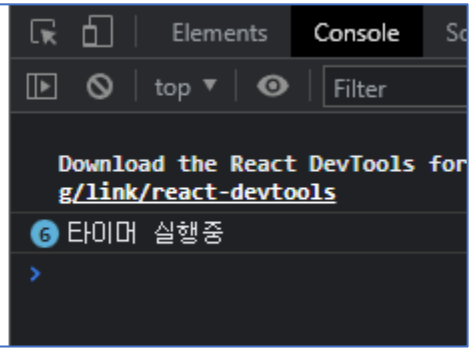
src/App.js

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!
- ExUnmount.jsx 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

useEffect

실전 활용!

실전 활용?

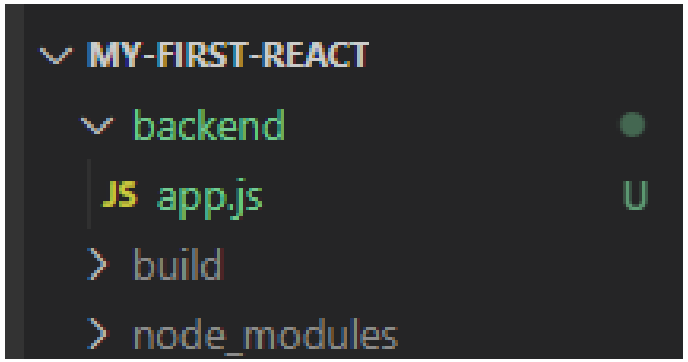
- useEffect 는 다양한 곳에서 활용이 가능합니다!
- 보통 컴포넌트가 서버로 부터 데이터를 받아와야 하는 상황에서 많이 사용
- 컴포넌트가 최초 마운트 → 서버로 부터 데이터를 요청 → 데이터를 State 에 등록 → 해당 내용을 렌더링
- 위와 같은 흐름을 많이 사용합니다!

간단한 백엔드 서버 구성하기!

- 지난번에 했었던 pororoObjArr 라는 배열을 전달해 주는 백엔드 서버를 간단하게 구축해 봅시다!
- 따로 만들 필요 까지는 없으니, 리액트 폴더에 backend 라는 폴더를 만들어서 간단하게 구현해 봅시다!
- 왜냐면 리액트는 이미 npm 이 관리하는 폴더이기 때문에 간단하게 필요 패키지 모듈만 설치하여 사용하면 되기 때문이죠!

간단한 백엔드 서버 구성하기!

- 일단 리액트 폴더에 backend 폴더 생성
- Express 와 cors 모듈 설치
- Npm i express cors



server.js 코드

```
const express = require("express");
const cors = require("cors");

const PORT = 4000;
const app = express();

app.use(cors());
app.get("/", (req, res) => {
  const pororoObjArr = [];
  res.send(JSON.stringify(pororoObjArr));
});

app.listen(PORT, () => {
  console.log(`데이터 통신 서버가 ${PORT}에서 작동 중입니다!`);
});
```

```
const pororoObjArr = [
    {
      name: "뽀로로",
      age: "5",
      nickName: "사고뭉치",
    },
    {
      name: "루피",
      age: "4",
      nickName: "공주님",
    },
    {
      name: "크롱이",
      age: "5",
      nickName: "장난꾸러기",
    },
  ];
```

간단한 백엔드 서버 구성하기!

- 서버 실행 후 Postman 으로 테스트

GET localhost:4000/

Params Authorization Headers (6) Body Pre-request Script Tests Se

Body Cookies Headers (8) Test Results

Pretty Raw Preview Visualize JSON


```
[
    {
        "name": "뽀로로",
        "age": "5",
        "nickName": "사고뭉치"
    },
    {
        "name": "루피",
        "age": "4",
        "nickName": "공주님"
    },
    {
        "name": "크롱이",
        "age": "5",
        "nickName": "장난꾸러기"
    }
]
```

리액트 컴포넌트 만들기!

- 백엔드 데이터를 받아서 그려줄 UseEffectFetch.jsx 컴포넌트 생성
- 함수가 마운트 되면 Fetch 함수를 이용해서 만들어 둔, API로 데이터를 요청하고 해당 데이터를 받아서 state 에 부여해 봅시다!

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

export default function UseEffectFetch() {
  const [dataArr, setDataArr] = useState([]);

  async function fetchData() {
    const resFetch = await fetch("http://localhost:4000/", {
      method: "GET",
      headers: {
        "Content-type": "application/json",
      },
    });
    const data = await resFetch.json();
    setDataArr(data);
    console.log(dataArr);
  }

  useEffect(() => {
    fetchData();
  }, []);
```

```
▼ (3) [{…}, {…}, {…}]
  ▶ 0: {name: '뽀로로', age: '5', nickName: '사고뭉치'}
  ▶ 1: {name: '루피', age: '4', nickName: '공주님'}
  ▶ 2: {name: '크롱이', age: '5', nickName: '장난꾸러기'}
    length: 3
  ▶ [[Prototype]]: Array(0)
```

받은 데이터를 그려 봅시다!

- 그런데 데이터가 일정하죠? 그럼 하나하나 그릴 필요가 없겠죠?
- 바로 컴포넌트와 props 를 활용해 봅시다!
- 데이터를 그려주는 ProfileComponent.jsx 를 생성해 봅시다!

```jsx
export default function ProfileComponent({ name, age, nickName }) {
  return (
    <div>
      <h1>이름 : {name}</h1>
      <h1>나이 : {age}</h1>
      <h1>별명 : {nickName}</h1>
      <hr />
    </div>
  );
}
```

컴포넌트를 임포트 하고 map 으로 그려주기!

- 생성한 ProfileComponent 컴포넌트를 임포트 하고, map 을 이용하여 그려 줍시다!

```
import ProfileComponent from "./ProfileComponent";
// 기존 코드

return (
    <>
      {dataArr.map((el) => {
        return (
          <ProfileComponent
            key={el.name}
            name={el.name}
            age={el.age}
            nickName={el.nickName}
          />
        );
      })}
    </>
  );
```

이름 : 뽀로로

나이 : 5

별명 : 사고뭉치

이름 : 루피

나이 : 4

별명 : 공주님

이름 : 크롱이

나이 : 5

별명 : 장난꾸러기


세상에서 제일 중요한 거거든
useState
useRef
useEffect


수고하셨습니다!