**1장. 리액트 입문1/4: 기초**

**<<01. 리액트는 어쩌다 만들어졌을까?>>**

\* JavaScript를 사용하여 HTML로 구성한 UI를 제어한다면, DOM을 변형시키기 위하여 브라우저의 DOM Selector API를 사용해서 특정 DOM을 선택한 뒤, 특정 이벤트가 발생하면 변화를 주도록 설정해야 함

\* HTML/JS로 만들어진 카운터를 예로 든다면:

- 사용자와의 인터랙션이 별로 없는 웹페이지라면 상관없겠지만, 만약에 인터랙션이 자주 발생하고, 이에 따라 동적으로 UI 를 표현해야된다면, 이러한 규칙이 정말 다양해질것이고, 그러면 관리하기도 힘들어질것입니다.

- 대부분의 경우 웹 애플리케이션의 규모가 커지면, DOM 을 직접 건드리면서 작업을 하면 코드가 난잡해지기 쉽습니다.

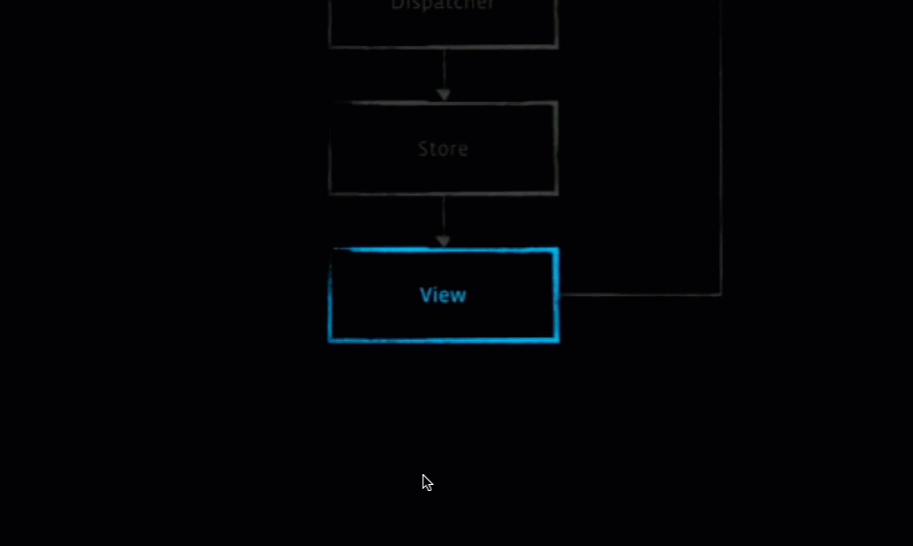
- 처리해야 할 이벤트도 다양해지고, 관리해야 할 상태값도 다양해지고, DOM 도 다양해지게 된다면, 이에 따라 업데이트를 하는 규칙도 많이 복잡해지기 때문에, 조금 과장을 많이 하자면 코드가 다음과 같은 형태가 됩니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

\* 그래서 Ember, Backbone, AngularJS등의 프레임워크가 만들어졌는데, 이 프레임워크들의 작동방식을 쉽게 설명하자면 자바스크립트의 특정값이 바뀌면 DOM의 속성을 바뀌도록 연결을 해주어서, 업데이트 하는 작업을 간소화해주는 방식으로 웹개발의 어려움을 해결해주었음

\* 하지만 리액트의 경우에는 조금 다른 발상에서 만들어짐. 리액트는 어떠한 상태가 바뀌었을 때, 그 상태에 따라 DOM을 어떻게 업데이트할지 규칙을 정하는 것이 아니라, 아예 다 날려버리고 처음부터 모든 걸 새로 만들어서 보여준다면 어떨까? 라는 아이디어에서 개발이 시작되었음



\* “업데이트를 어떻게 해야 할 지”에 대한 고민을 전혀 안 해도 되기 때문에 개발이 정말 쉬어질 것임. 하지만 매우 동적인 UI를 보여주기 위해서 모든 걸 다 날려버리고 처음부터 새롭게 만들게 된다면, 속도가 굉장히 느릴 것임. 특히 규모가 큰 웹 애플리케이션이라면 상상도 할 수 없는 일임.

\* 하지만 **리액트**는 Virtual DOM이라는 것을 사용해서 이를 가능케 했음

\* **Virtual DOM**이란: 브라우저에서 실제로 보여지는 DOM이 아니라 그냥 메모리에 가상으로 존재하는 DOM으로서 그냥 JavaScript 객체이기 때문에 작동 성능이 실제로 브라우저에서 DOM을 보여주는 것보다 속도가 훨씬 빠름. 리액트는 상태가 업데이트 되면, 업데이트가 필요한 곳의 UI를 Virtual DOM을 통해서 렌더링함. 그리고나서 리액트 개발팀이 만든 매우 효율적인 비교 알고리즘을 통하여 실제 브라우저에 보여지고 있는 DOM과 비교를 한 후, 차이가 있는 곳을 감지하여 이를 실제 DOM에 패치 시켜줌. 이를 통해 “업데이트를 어떻게 할 지”에 대한 고민을 하지 않으면서, 빠른 성능도 지켜낼 수 있게 됨

**<<02. 작업환경 준비>>**

**\* 다음 항목들을 설치**

**- Node.js:** Webpack 과 Babel 같은 도구들이 자바스크립트 런타임인 Node.js 를 기반으로 만들어져있습니다. 그렇기에 해당 도구들을 사용하기 위해서 Node.js 를 설치합니다.

**- Yarn:** Yarn 은 조금 개선된 버전의 npm 이라고 생각하시면 됩니다. npm 은 Node.js 를 설치하게 될 때 같이 딸려오는 패키지 매니저 도구입니다. 프로젝트에서 사용되는 라이브러리를 설치하고 해당 라이브러리들의 버전 관리를 하게 될 때 사용하죠. 우리가 Yarn 을 사용하는 이유는, 더 나은 속도, 더 나은 캐싱 시스템을 사용하기 위함입니다.

**- 코드 에디터:** 그리고, 코드 에디터를 준비하세요. 여러분이 좋아하는 에디터가 있다면, 따로 새로 설치하지 않고 기존에 사용하시던걸 사용하셔도 됩니다. 저는 주로 VSCode 를 사용합니다. 이 외에도, Atom, WebStorm, Sublime 같은 훌륭한 선택지가 있습니다.

**- Git bash:** 윈도우의 경우, Git for Windows 를 설치해서 앞으로 터미널에 무엇을 입력하라는 내용이 있으면 함께 설치되는 Git Bash 를 사용하세요. 윈도우가 아니라면 설치하지 않으셔도 상관없습니다. 설치는 기본 옵션으로 진행하시면 됩니다.

**\* React APP 만드는 법:**

- Git Bash에 들어감

$ npx create-react-app begin-react

$ cd begin-react

$ npm start

\* C:\Users\dlatl\begin-react에 생성

**<<03. 나의 첫번째 리액트 컴포넌트>>**

**\* Hello.js**

import React from 'react';

function Hello() {

return <div>안녕하세요</div>

}

export default Hello;

- 리액트 컴포넌트를 만들 땐 import 를 통하여 리액트를 불러와주어야 함

- 리액트 컴포넌트에서는 XML 형식의 값을 변환해줄 수 있는데 이를 **JSX**라고 부름.

- export~ 이 코드는 Hello라는 컴포넌트를 내보내겠다는 의미임. 이렇게 해주면 다른 컴포넌트에서 불러와서 사용할 수 있음.

**\* App.js**

import React from 'react';

import Hello from './Hello';

function App() {

return (

<div>

<Hello />

<Hello />

<Hello />

</div>

);

}

export default App;

- **컴포넌트**는 일종의 UI 조각임. 그리고 쉽게 재사용할 수 있음

**\* index.js**

import React from 'react';

import ReactDOM from 'react-dom';

import './index.css';

import App from './App';

import \* as serviceWorker from './serviceWorker';

ReactDOM.render(<App />, document.getElementById('root'));

// If you want your app to work offline and load faster, you can change

// unregister() to register() below. Note this comes with some pitfalls.

// Learn more about service workers: https://bit.ly/CRA-PWA

serviceWorker.unregister();

- 여기서 **ReactDOM.render**의 역할은 브라우저에 있는 실제 DOM 내부에 리액트 컴포넌트를 렌더링하겠다는 것을 의미합니다. Id가 root인 DOM을 선택하고 있는데, 이 DOM이 어디있는지 볼까요?

**\* public/index.html**을 열어보면 내부에

<div id="root"></div>

을 찾아볼 수 있음.

- 결국 리액트 컴포넌트가 렌더링 될 때에는 렌더링된 결과물이 위 div 내부에 렌더링되는 것임

**<<04. JSX의 기본 규칙 알아보기>>**

**\* JSX:** 리액트의 생김새를 정의할 때 사용하는 문법. HTML처럼 생겼지만 실제로는 JavaScript임.

**\* Babel:**

- 자바스크립트의 문법을 확장해주는 도구임

- 아직 지원되지 않는 최신 문법이나, 편의상 사용하거나 실험적인 자바스크립트 문법들을 정식 자바스크립트 형태로 변환해줌으로서 구형 브라우저 같은 환경에서도 제대로 실행할 수 있게 해주는 역할을 함

- 리액트 컴포넌트 파일에서 XML 형태로 코드를 작성하면 babel이 JSX를 JavaScript로 변환해줌

텍스트이(가) 표시된 사진

자동 생성된 설명

**\* JSX가 JavaScript로 제대로 변환되려면 지켜야하는 규칙:**

import React from 'react';

import Hello from './Hello';

function App() {

return (

<div>

<Hello />

<Hello />

<Hello />

<input /> //Self Closing 태그

<br />

</div>

);

}

export default App;

**- 꼭 닫혀있어야 하는 태그:** HTML에서는 input 또는 br태그를 사용할 때 닫지 않고 사용하기도 함. 하지만 리액트에서는 그렇게 하면 안 됨.

- 태그와 태그 사이에 내용이 들어가지 않을 때는 Self Closing 태그 라는 것을 사용해야 함. 현재 Hello 컴포넌트를 사용할 때에도 Self Closing 태그를 사용해주었는데, 열리고 바로 닫히는 태그를 의미함

import React from 'react';

import Hello from './Hello';

function App() {

return (

<> //fragment 사용

<Hello />

<div>안녕히계세요</div>

</>

);

}

export default App;

- **꼭 감싸져야하는 태그:** 두 개 이상의 태그는 무조건 하나의 태그로 감싸져있어야 함

- 단순히 감싸기 위하여 불필요한 div 로 감싸는게 별로 좋지 않은 상황도 있습니다. 예를 들어서 스타일 관련 설정을 하다가 복잡해지게 되는 상황도 올 수 있고, table 관련 태그를 작성 할 때에도 내용을 div 같은걸로 감싸기엔 애매하죠. 그럴 땐, 리액트의 **Fragment** 라는 것을 사용하면 됩니다.

ㄴ **Fragment**: 태그를 작성할 때 이름 없이 작성을 하게 되면 Fragment가 만들어지는데, Fragment는 브라우저 상에서 따로 별도의 엘리먼트로 나타나지 않음

**\* JSX 안에 자바스크립트 값 사용하기**

- JSX 내부에 자바스크립트 변수를 보여줘야 할 때는 {} 으로 감싸서 보여줌

import React from 'react';

import Hello from './Hello';

function App() {

const name = 'react';

return (

<>

<Hello />

<div>{name}</div> //{} 사용

</>

);

}

export default App;

**\* styel과 className**

import React from 'react';

import Hello from './Hello';

function App() {

const name = 'react';

const style = {

backgroundColor: 'black',

color: 'aqua',

fontSize: 24, // 기본 단위 px

padding: '1rem' // 다른 단위 사용 시 문자열로 설정

}

return (

<>

<Hello />

<div style={style}>{name}</div>

</>

);

}

export default App;

- JSX에서 태그에 style과 CSS class를 설정하는 방법을 HTMl에서 설정하는 방법과 다름

- 인라인 스타일은 객체 형태로 작성해야 하며, background-color처럼 -로 구분되어 있는 이름들은 backgroundColor처럼 camelCase형태로 네이밍 해주어야 함

- CSS class를 설정할 때에는 class= 가 아닌 className= 으로 설정해주어야 함

**\* App.css**

.gray-box {

background: gray;

width: 64px;

height: 64px;

}

**\* App.js**

import React from 'react';

import Hello from './Hello';

import './App.css';

function App() {

const name = 'react';

const style = {

backgroundColor: 'black',

color: 'aqua',

fontSize: 24, // 기본 단위 px

padding: '1rem' // 다른 단위 사용 시 문자열로 설정

}

return (

<>

<Hello />

<div style={style}>{name}</div>

<div className="gray-box"></div>

</>

);

}

export default App;

**\* 결과**

텍스트이(가) 표시된 사진

자동 생성된 설명

**\* 주석:** JSX 내부의 주석은 {/\* 이런 형태로 \*/} 작성함

<>

{/\* 주석은 화면에 보이지 않습니다 \*/}

/\* 중괄호로 감싸지 않으면 화면에 보입니다 \*/

<Hello />

<div style={style}>{name}</div>

<div className="gray-box"></div>

</>

텍스트이(가) 표시된 사진

자동 생성된 설명

**\* 열리는 태그 주석:** 열리는 태그 내부에서는 //이런 형태로 주석 작성이 가능함

<>

{/\* 주석은 화면에 보이지 않습니다 \*/}

/\* 중괄호로 감싸지 않으면 화면에 보입니다 \*/

<Hello //열리는 태그 내부에서는 이렇게 주석을 작성할 수 있습니다

/>

<div style={style}>{name}</div>

<div className="gray-box"></div>

</>

**<<05.props를 통해 컴포넌트에게 값 전달하기>>**

\* props: 컴포넌트의 props는 properties의 줄임말, 어떠한 값을 컴포넌트에게 전달해줘야 할 때 사용

\* props의 기본 사용법

- App 컴포넌트에서 Hello 컴포넌트를 사용할 때 name 이라는 값을 전달해주고 싶다 가정

- App.js

import React from 'react';

import Hello from './Hello';

function App() {

return (

<Hello name="react" /> //전달하고 싶은 name값

);

}

export default App;

- Hello.js

import React from 'react';

function Hello(props) { //파라미터로 props 받기

return <div>안녕하세요 {props.name}</div> //{props.값이름} 으로 받음

}

export default Hello;

- 컴포넌트에게 전달되는 props는 파라미터를 통하여 조회할 수 있음. Props는 객체 형태로 전달되며, 만약 name 값을 조회하고 싶다면 props.name을 조회하면 됨

**\* 여러 개의 props**

- App.js

import React from 'react';

import Hello from './Hello';

function App() {

return (

<Hello name="react" color="red"/> //전달하고 싶은 name값과 color값

);

}

export default App;

-Hello.js

import React from 'react';

function Hello(props) {

return <div style={{ color: props.color }}>안녕하세요 {props.name}</div>

}

export default Hello;

**\* 비구조화 할당**

\* props내부의 값을 조회할 때마다 props.를 사용하고 있는데, 함수의 파라미터에서 비구조화 할당(구조 분해) 문법을 사용하면 코드를 간결하게 작성할 수 있음

- Hello.js

import React from 'react';

function Hello({ color, name }) {

return <div style={{ color }}>안녕하세요 {name}</div>

}

export default Hello;

\* defaultProps로 기본값 설정

- 컴포넌트에 props를 지정하지 않았을 때 기본적으로 사용할 값을 설정하고 싶다면 컴포넌트에 defaultProps라는 값을 설정하면 됨

- Hello.js

import React from 'react';

function Hello({ color, name }) {

return <div style={{ color }}>안녕하세요 {name}</div>

}

Hello.defaultProps = { //defaultProps 설정

name: '이름없음'

}

export default Hello;

- App.js

import React from 'react';

import Hello from './Hello';

function App() {

return (

<>

<Hello name="react" color="red"/>

<Hello color="pink"/>

</>

);

}

export default App;

**\* props.children:** 컴포넌트 태그 사이에 넣은 값을 조회하고 싶을 때

- props.children을 사용하는 새로운 컴포넌트를 만들겠음

- Wrapper.js

import React from 'react';

function Wrapper() {

const style = {

border: '2px solid black',

padding: '16px',

};

return (

<div style={style}>

</div>

)

}

export default Wrapper;

- App.js

import React from 'react';

import Hello from './Hello';

import Wrapper from './Wrapper';

function App() {

return (

<Wrapper> //Wrapper

<Hello name="react" color="red"/>

<Hello color="pink"/>

</Wrapper>

);

}

export default App;

- 결과값

텍스트이(가) 표시된 사진

자동 생성된 설명

ㄴ 내부의 내용을 보여지게 하기 위해서는 Wrapper에서 props.children을 렌더링해줘야 함

- Wrapper.js

import React from 'react';

function Wrapper({ children }) { //파라미터로 children 전달

const style = {

border: '2px solid black',

padding: '16px',

};

return (

<div style={style}>

{children} //children 렌더링

</div>

)

}

export default Wrapper;

텍스트이(가) 표시된 사진

자동 생성된 설명

**<<06. 조건부 렌더링>>**

\* 조건부 렌더링: 특정 조건에 다라 다른 결과물을 렌더링하는 것

\* isSpecial의 값을 통해 렌더링

- App.js

import React from 'react';

import Hello from './Hello';

import Wrapper from './Wrapper';

function App() {

return (

<Wrapper>

<Hello name="react" color="red" isSpecial={true}/>

<Hello color="pink" />

</Wrapper>

)

}

export default App;

- Hello.js

import React from 'react';

function Hello({ color, name, isSpecial }) { //파라미터로 isSpecial 받음

return (

<div style={{ color }}>

{ isSpecial ? <b>\*</b> : null }

안녕하세요 {name}

</div>

);

}

Hello.defaultProps = {

name: '이름없음'

}

export default Hello;

텍스트이(가) 표시된 사진

자동 생성된 설명

\* 단축 평가 논리 계산법을 이용한 조건부 렌더링

- Hello.js

import React from 'react';

function Hello({ color, name, isSpecial }) {

return (

<div style={{ color }}>

{isSpecial && <b>\*</b>} //isSpecial false => false, true => <b>\*</b>

안녕하세요 {name}

</div>

);

}

Hello.defaultProps = {

name: '이름없음'

}

export default Hello;

**\* props값 설정을 생략하면: ={true}**

**-** isSpecial 이름만 넣어주면 isSpecial={true} 와 동일한 의미

- App.js

import React from 'react';

import Hello from './Hello';

import Wrapper from './Wrapper';

function App() {

return (

<Wrapper>

<Hello name="react" color="red" isSpecial /> //기본값으로 true가 들어감

<Hello color="pink"/>

</Wrapper>

);

}

**<<07.useState를 통해 컴포넌트에서 바뀌는 값 관리하기>>**

\* 컴포넌트에서 보여줘야 하는 내용이 사용자 인터랙션에 따라 바뀌어야 할 때 어떻게 구현할 수 있는지에 대해 알아볼 것임

**\* 리액트 16.8이전 버전:** 함수형 컴포넌트에서는 상태를 관리할 수 없었음

**\* 리액트 16.8:** Hooks라는 기능이 도입되면서 함수형 컴포넌트에서도 상태를 관리할 수 있게 되었음.

**\* useState 함수**: 리액트의 Hooks 중 하나인 useState 함수

**\* Counter 조정**

- Counter.js

import React from 'react';

function Counter() {

return (

<div>

<h1>0</h1>

<button>+1</button>

<button>-1</button>

</div>

);

}

export default Counter;

- App.js

import React from 'react';

import Counter from './Counter';

function App() {

return (

<Counter />

);

}

export default App;

- 결과

텍스트이(가) 표시된 사진

자동 생성된 설명

**\* 이벤트 설정:** Counter에서 버튼이 클릭되는 이벤트가 발생했을 때 특정 함수가 호출되도록 설정

- Counter.js

import React from 'react';

function Counter() {

const onIncrease = () => { //+ 이벤트 함수 작성

console.log('+1')

}

const onDecrease = () => { //- 이벤트 함수 작성

console.log('-1');

}

return (

<div>

<h1>0</h1>

<button onClick={onIncrease}>+1</button> //+ 이벤트 함수 설정

<button onClick={onDecrease}>-1</button> //- 이벤트 함수 설졍

</div>

);

}

export default Counter;

**\* 동적인 값 끼얹기, useState:**

- 컴포넌트에서 동적인 값을 상태(state)라고 부름, 리액트에는 useState라는 함수가 있는데 이것을 사용하면 컴포넌트에서 상태를 관리할 수 있음

- Counter.js

import React, { useState } from 'react'; //리액트 패키지에서 useState라는 함수 불러와줌

function Counter() {

const [number, setNumber] = useState(0); //useState를 사용할 때에는 상태의 기본값을 파라미터로 넣어서 호출해줌. 이 함수를 호출해주면 배열이 반환되는데, 여기서 첫번째 원소는 현재 상태, 두번째 원소는 Setter 함수임

const onIncrease = () => {

setNumber(number + 1); //Setter함수는 파라미터로 전달 받은 값을 최신 상태로 설정해줌

}

const onDecrease = () => {

setNumber(number - 1); //Setter함수는 파라미터로 전달 받은 값을 최신 상태로 설정해줌

}

return (

<div>

<h1>{number}</h1>

<button onClick={onIncrease}>+1</button>

<button onClick={onDecrease}>-1</button>

</div>

);

}

export default Counter;

\* 배열 비구조화 할당 활용

const [number, setNumber] = useState(0);

와

const numberState = useState(0);

const number = numberState[0];

const setNumber = numberState[1];

는 동일함. 배열 비구조화 할당을 통하여 각 원소를 추출해준 것임

**\* 함수형 업데이터:** 기존값을 어떻게 업데이트할지에 대한 함수를 등록하는 방식으로도 값을 업데이트 할 수 있음

- 나중에 컴포넌트를 최적화하게 될 때 사용함

- Counter.js

import React, { useState } from 'react';

function Counter() {

const [number, setNumber] = useState(0);

const onIncrease = () => { //값을 업데이트 하는 함수를 파라미터로 넣어줌

setNumber(prevNumber => prevNumber + 1);

}

const onDecrease = () => {

setNumber(prevNumber => prevNumber - 1);

}

return (

<div>

<h1>{number}</h1>

<button onClick={onIncrease}>+1</button>

<button onClick={onDecrease}>-1</button>

</div>

);

}

export default Counter;

**<<08.input 상태 관리하기>>**

**\* input의 onChange 이벤트:** 이벤트에 등록하는 함수에서 이벤트 객체 e를 파라미터로 받아와서 사용할 수 있는데, 이 객체의 e.target은 이벤트가 발생한 DOM인 input DOM을 가리키게 됨. 이 DOM의 value값, 즉 e.target.value를 조회하면 현재 input에 입력한 값이 무엇인지 알 수 있음

- 해당 값을 useState를 통해서 관리해주면 됨

- Inputsample.js

import React, { useState } from 'react';

function InputSample() {

const [text, setText] = useState('');

const onChange = (e) => {

setText(e.target.value);

};

const onReset = () => {

setText('');

};

return (

<div>

<input onChange={onChange} value={text} />

<button onClick={onReset}>초기화</button>

<div>

<b>값: {text}</b>

</div>

</div>

);

}

export default InputSample;

**<<09. 여러 개의 input 상태 관리하기>>**

**\* input 개수가 여러 개일 때:**

- 단순히 useState를 여러 번 사용하고 onChange도 여러 개 만들어서 구현할 수 있지만 좋은 방법은 아님

- input에 name을 설정하고 이벤트가 발생했을 때 이 값을 참조하는 것이 좋음

- useState에서는 문자열이 아니라 객체 형태의 상태를 관리해주어야 함

- InputSample.js

import React, { useState } from 'react';

function InputSample() {

const [inputs, setInputs] = useState({

name: '',

nickname: ''

});

const { name, nickname } = inputs; // 비구조화 할당을 통해 값 추출

const onChange = (e) => {

const { value, name } = e.target; // 우선 e.target 에서 name 과 value 를 추출

setInputs({

...inputs, // 기존의 input 객체를 복사한 뒤

[name]: value // name 키를 가진 값을 value 로 설정

});

};

const onReset = () => {

setInputs({

name: '',

nickname: '',

})

};

return (

<div>

<input name="name" placeholder="이름" onChange={onChange} value={name} />

<input name="nickname" placeholder="닉네임" onChange={onChange} value={nickname}/>

<button onClick={onReset}>초기화</button>

<div>

<b>값: </b>

{name} ({nickname})

</div>

</div>

);

}

export default InputSample;

**\* 리액트 상태에서 객체를 수정해야 할 때**

inputs[name] = value;

- 이런 식으로 하면 안 됨

- 새로운 객체를 만들어 새로운 객체에 변화를 주고, 이 상태로 사용해주어야 함

setInputs({

...inputs,

[name]: value

});

- “불변성을 지킨다”: 이러한 작업을 “불변성을 지킨다”라고 부름. 불변성을 지켜주어야만 리액트 컴포넌트에서 상태가 업데이트가 됐음을 감지할 수 있고 이에 따라 필요한 리렌더링이 진행됨. 만약 inputs[name] = value; 와 같이 기존 상태를 직접 수정하게 되면 값을 바꾸도 리렌더링 되지 않음

ㄴ 그 이유: 리액트 컴포넌트가 re-rendering을 할 때, virtual DOM에서 이전 컴포넌트와 새로 바뀌는 컴포넌트를 비교하게 될텐데 기존 객체를 직접 수정하게 되면 새로 바뀐 컴포넌트에서는 이전 컴포넌트와 같은 상태를 가질 수 있다고 보아 "새 객체"에 변화를 주어야 한다고 말하는 것 같습니다.

- 결론 : 리액트에서 객체를 업데이트하게 될 때에는 기존 객체를 직접 수정하면 안되고, 새로운 객체를 만들어서, 새 객체에 변화를 주어야 됩니다.

**\* 최적화:** 리액트에서는 불변성을 지켜주어야만 컴포넌트 업데이트 성능 최적화를 제대로 할 수 있음

**<<10. useRef로 특정 DOM 선택하기>>**

**\* ref:**

- 리액트를 사용하는 프로젝트에서도 가끔씩 DOM 을 직접 선택해야 하는 상황이 발생 할 때도 있습니다. 예를 들어서 특정 엘리먼트의 크기를 가져와야 한다던지, 스크롤바 위치를 가져오거나 설정해야된다던지, 또는 포커스를 설정해줘야된다던지 등 정말 다양한 상황이 있겠죠. 추가적으로 Video.js, JWPlayer 같은 HTML5 Video 관련 라이브러리, 또는 D3, chart.js 같은 그래프 관련 라이브러리 등의 외부 라이브러리를 사용해야 할 때에도 특정 DOM 에다 적용하기 때문에 DOM 을 선택해야 하는 상황이 발생 할 수 있습니다.

- 그럴 땐 리액트에서 ref라는 것을 사용함

**\* useRef:** 함수형 컴포넌트에서 ref를 사용할 때에는 useRef라는 Hook함수를 사용함. 클래스형 컴포넌트에서는 콜백 함수를 사용하거나 React.createRef라는 함수를 사용함

**\* useRef 사용:** InputSample에서는 초기화 버튼을 누르면 포커스가 초기화 버튼에 그대로 남아있게 됨. 초기화 버튼을 클릭했을 때 이름 input에 포커스가 잡히도록 useRef를 사용하여 기능을 구현

- InputSample.js

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

function InputSample() {

const [inputs, setInputs] = useState({

name: '',

nickname: ''

});

const nameInput = useRef();

const { name, nickname } = inputs; // 비구조화 할당을 통해 값 추출

const onChange = e => {

const { value, name } = e.target; // 우선 e.target 에서 name 과 value 를 추출

setInputs({

...inputs, // 기존의 input 객체를 복사한 뒤

[name]: value // name 키를 가진 값을 value 로 설정

});

};

const onReset = () => {

setInputs({

name: '',

nickname: ''

});

nameInput.current.focus();

};

return (

<div>

<input

name="name"

placeholder="이름"

onChange={onChange}

value={name}

ref={nameInput}

/>

<input

name="nickname"

placeholder="닉네임"

onChange={onChange}

value={nickname}

/>

<button onClick={onReset}>초기화</button>

<div>

<b>값: </b>

{name} ({nickname})

</div>

</div>

);

}

export default InputSample;