00.

나는 비전공자이며, 프론트엔드로 방향을 정한지 얼마되지 않아 이것저것 알아보았을 때, 리액트를 많이 쓴다는 것을 알게 되었다. 그래서 '무엇을 이용해서 지식을 얻어야할까?' 하고 고민을 했다. 그러다 한 가지 책을 추천받았는데 그 책은 바로`리액트를 다루는 기술`이다.

책에 대한 내용 정리는 벨로그에서 작성하고, 깃허브에 올릴 예정이다.

아직은 초반이라 그런지는 몰라도 책에 대한 거부감도 없을 뿐더러 오히려 책이 재밌어서 공부하기에 괜찮은 것 같다.

> ### 나한테 리액트에 대한 개념 이해도가 왜 필요할까?

처음에 말했듯이 나는 비전공자라 나는 먼저 내가 이것을 왜 쓰는지, 어떻게 쓰는지 전혀 모른다.

내가 이것을 사용하는 이유를 알아야 더 재밌어지고, 사용하는 데에 있어서 거부감이 들지 않기 때문에 가장 먼저 리액트를 사용하는 이유를 알아야할 것 같다.

01.

> ### 리액트

- 최대한 성능을 아끼고 편안한 사용자 경험을 제공하면서 구현하고자 개발

- 자바스크립트 라이브러리로 사용자 인터페이스를 만드는데 사용

- \*\*오직 뷰(View)만 신경 쓰는 라이브러리\*\*이다.

(기타 기능은 직접 구현하여 사용해야 한다.)

- 따라서, \*\*라우팅에는 리액트 라우터\*\*, \*\*Ajax 처리는 axios나 fetch\*\*, \*\*상태 관리는 리덕스, MobX를 사용\*\*하여 빈자리를 채우면 된다.

- 여러 라이브러리를 접해야 한다는 단점이 있지만, 자신의 취향대로 스택을 설정할 수 있다는 장점과 다른 웹 프레임워크나 라이브러리와 혼용할 수도 있다.

\*\*주의 할 점\*\*

- 리액트를 사용하기 전에 작업이 매우 간단할 때는 사용하지 않는 편이 더 나은 성능을 보이기도 한다.

---

리액트 프로젝트에서 특정 부분이 어떻게 생길지 정하는 선언체가 있는데, 이를 \*\*컴포넌트\*\*라고 한다.

<br>

> ### \*\* 컴포넌트 \*\*

- 재사용이 가능한 API로 수많은 기능들을 내장

- 컴포넌트 하나에서 해당 컴포넌트의 생김새와 작동 방식을 정의.

데이터가 변할 때마다 새롭게 리렌더링하면서 성능을 아끼고, 최적의 사용자 경험을 제공할 수 있는 이유는 무엇일까?

이에 대한 답은 \*\*'초기 렌더링'\*\*과 \*\*'리렌더링'\*\* 개념을 이해해야 한다고 한다.

UI관련 프레임워크, 라이브러리를 사용하든지 간에 맨 처음 보일지를 정하는 초기 렌더링이 필요한데 리액트에서는 \*\*render 함수\*\*가 있다.

>### \*\*render 함수\*\*

- 컴포넌트가 어떻게 생격는지 정의

- html 형식의 문자열을 반환하지 않음

- 뷰가 어떻게 생겼고 어떻게 작동하는지에 대한 정보를 지닌 객체를 반환

<br>

컴포넌트는 데이터를 업데이트했을 때 단순히 업데이트한 값을 수정하는 것이 아니라, 새로운 데이터를 가지고 render 함수를 또 다시 호출한다.

이때 데이터를 지닌 뷰를 생성하게 되는데 render 함수는 반환하는 결과를 곧바로 DOM에 반영하지 않고, 이전에 만들었던 컴포넌트 정보와 비교한 후 둘의 차이를 알아내어 최소한의 연산으로 DOM 트리를 업데이트 하는 것이다.

전체 컴포넌트를 다시 렌더링하는 것처럼 보이지만, 최적의 사원을 사용하여 이를 수행하는 것이다.

---

컴포넌트 내부에는 또 다른 컴포넌트들이 들어갈 수 있다. 이때 render 함수를 실행하면 내부에 있는 컴포넌트 들도 재귀적으로 렌더링한다.

이렇게 최상위 컴포넌트의 렌더링 작업이 끝나면 지니고 있는 정보들을 사용하여 HTML 마크업을 만들고 내가 정하고자 하는 실제 페이지의 DOM 요소 안에 주입한다.

>렌더링 절차(컴포넌트를 실제 페이지에 렌더링 할 때)

1. 문자열 형태의 HTML 코드를 생성

2. 특정 DOM에 해당 내용을 주입하면 이벤트가 적용됨.

---

구체적으로 이해하기 위해서는 Virtual DOM을 알아야 한다고 해서 그 다음장에 있는 내용을 읽게 되었다.

> ### DOM

- Document Object Model의 약어로 \*\*객체로 문서 구조를 표현하는 방법\*\*

(방법으로는 XML이나 HTML로 작성한다.)

- 트리 형태라서 특정 노드를 찾거나 제거하거나 원하는 곳에 삽입할 수 있다.<br>

DOM API를 수많은 플랫폼과 웹 브라우저에서 사용하는 편이지만,

DOM의 치명적인 단점으로 \*\*종적 UI에 최적화되어 있지 않다는 것\*\*이다.

HTML은 자체적으로는 정적이지만, 자바스크립트를 사용하여 이를 동적으로 만들 수 있다.

규모가 큰 퀩 애플리케이션에서 DOM에 직접 접근하여 변화를 주다보면 느려지기 시작한다.

이를 두고 'DOM은 느리다' 라고 하지만, DOM 자체는 빠르다 대신 웹 브라우저가 CSS를 다시 연산하고, 레이아웃을 구성하고, 페이지를 리페인트 하는 과정에서 시간을 허비하게 되기 때문에 느리다고 표현을 하는 것이다.

DOM을 최소한으로 조작하여 작업을 처리하는 방식으로 개선할 수 있다고 한다.

여기서 리액트는 Virtual DOM 방식을 사용하여 DOM 업데이트를 추상화함으로써 DOM 처리 횟수를 최소하 하고 효율적으로 진행한다.

> ### Virtual DOM

리액트의 주요 특징 중 하나는 Virtual DOM을 사용하는 것이라고 한다.<br>

이것을 사용하면 실제 DOM에 접근하여 조작하는 대신, 이를 추상화한 자바스크립트 객체를 구성하여 사용한다고 한다.

리액트에서 데이터가 변하여 웹브라우저에 실제 DOM을 업데이트할 때는 세 가지 절차가 진행되는데 절차 과정은 아래와 같다.

1. 데이터를 업데이트하면 전체UI를 Virtual DOM에 리렌더링한다.

2. 이전 DOM에 있던 내용과 현재 내용을 비교한다

3. 바뀐 부분만 실제 DOM에 적용한다.

---

리액트와 Virtual DOM이 언제나 제공하는 것은 \*\*업데이트 처리 간결성\*\*이다.

2장

> ### 번들러 (웹팩)

- 리액트 프로젝트에서는 주로 \*\*웹팩\*\*을 사용하는 추세이다.

(편의성과 확장성이 다른 도구보다 뛰어나기 때문.)

- 모듈을 불러왔을 때 불러온 모듈을 모두 합쳐서 하나의 파일을 생성해준다.

- SVG 파일과 CSS 파일도 불러와서 사용 가능. - \*\*로더 기능 담당.\*\*

- \*\*css, file - loader\*\*는 각각 CSS 파일, 웹 폰트나 미디어 파일 등을 불러오며

- \*\*babel-loader\*\*는 자바스크립트 파일들을 불러오면서 최신 JS 문법으로 작성된 코드를 바벨이라는 도구를 사용하여 ES5 문법으로 변환해줌

(구 버전 웹 브라우저와 호환하기 위해서 ES5로 변환해주어야 한다.)

종류로는 웹팩, Parcel, browserify라는 도구가 있음. (도구마다 특성이 다르다.)

---

> ### JSX

- 자바스크립트의 확장 문법, XML과 매우 비슷하게 생겼다.

- JSX 안에서는 자바스크립트 표현식을 쓸 수 있다. 내부에서 코드를 { }로 감싸면 된다.

- 보기 쉽고 익숙하며, 높은 활용도를 가지고 있음.

- JSX를 여러 줄로 작성할 때 괄호로 감싸고, 한 줄로 표현할 수 있는 JSX는 감싸지 않는다.

### JSX 문법

1. 컴포넌트에 여러 요소가 있다면 반드시 부모 요소 하나로 감싸야한다.

- Virtual DOM에서 컴포넌트 변화를 감지해 낼 때 효율적으로 비교할 수 있도록 \*\*컴포넌트 내부는 하나의 DOM 트리 구조로 이루어져야 한다\*\*는 규칙이 있기 때문.

>```

function App() {

return (

<div> or <Fragment> or <>

<h1> 리액트 ! </h1>

<h2>작동하나요?</h2>

</div> or </Fragment> or </>

);

}

exprot default App;

```

`여기서 <div>와 같이 감싸져 있어야한다. `

`(div 요소를 사용하고 싶지 않다면 Fragment, <>을 사용해도 됨.)`

2. if 문 대신 조건부 연산자(삼항 연산자)를 사용해야 한다.

- 조건에 따라 다른 내용을 렌더링해야 할 때

2-1. JSX 밖에서 if 문을 사용하여 사전에 값을 설정

2-2. { } 안에 조건부 연산자를 사용

`(삼항 연산자 : 조건문 ? 참 값 : 거짓 값)`

>```

function App() {

const name = '리액트';

return(

<div>

{name === '리액트' ? (<h1>리액트 </h1>) : (<h2> 리액트가 아님 </h2>)}

</div>

);

}

exprot default App;

```

2-3. 특정 조건을 만족할 때만 렌더링할 때는 조건부 연산자에 AND 연산자(&&)를 사용한다.

`조건문 && 참 값`

> ```

function App() {

const name = 'React';

return(

<div>

{name === '리액트' && <h1>리액트 </h1>}

</div>

);

}

exprot default App;

```

`여기서 name은 한글로 리액트가 아닌 React 이므로 결과는 출력되지 않는다.`

3. 함수에서 undefined만 반환하여 렌더링하는 상황을 만들면 안된다.

> ```

import './App.css';

function App() {

const name = undefeined;

return name;

}

exprot default App;

```

`이러한 상황은 오류를 발생한다.`

따라서,

```

import './App.css';

function App() {

const name = undefeined;

return name || '값이 undefined입니다.';

}

exprot default App;

```

` OR 연산자를 사용하여 || '값이 undefined입니다.' 와 같이 값을 지정하면 오류를 방지할 수 있다.`

```

return <div>{name}</div>

```

`또한, return 값을 위 코드 처럼 JSX 내부에서 undefined를 렌더링 하는 것은 괜찮다.`

4. 리액트에서 DOM 요소에 스타일을 적용할 때는 문자열 형태로 넣는 것이 아니라 객체 형태로 넣어 주어야 한다.

>```

function App() {

const name = '리액트';

const style = {

backgroundColor: 'black'

color: 'aqua',

fontSize: '48px',

fontWeight: 'bold',

padding: 16 //단위를 생략하면 px로 지정됨.

};

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

}

exprot default App;

```

`background-color는 backgorundColor와 같이 - 문자를 없애고 카멜 표기법으로 작성된 것을 볼 수 있다.`

`추가로 font-size, font-weight도 카멜 표기법으로 작성되었다.`

미리 선언하지 않고 바로 style 값을 지정하고 싶을 경우에는

>```

function App() {

const name = '리액트';

return <div style=

backgroundColor: 'black'

color: 'aqua',

fontSize: '48px',

fontWeight: 'bold',

padding: 16 >

{name}

</div>;

}

exprot default App;

```

`이렇게 표현할 수 있다.`

5. JSX에서는 className으로 설정해주어야 한다.

- 일반 HTML에서 CSS 클래스를 사용할 때는 ```<div class="myclass"></div>```라는 속성을 설정한다.

> css 클래스를 작성한 후 아래 코드를 입력하고 저장한다.

```

.react {

backgroundColor: 'black'

color: 'aqua',

fontSize: '48px',

fontWeight: 'bold',

padding: 16

}

```

이것을 js파일에 적용하려면

```

import './App.css';

function App() {

const name = '리액트';

return <div className ="react">{name}</div>;

}

export default App;

```

`className`처럼 작성해주면 된다.

`(class로 값을 설정해도 적용되긴 하지만 Console 탭에 경고 문구가 띄워진다.)`

6. 태그는 닫아주어야 한다.

> - 태그 사이에 별도의 내용이 들어가지 않는 경우 self-closing 태그를 이용하여 선언과 동시에 닫는 태그를 적용한다 `ex) <input />`

7. 주석

> - JSX의 주석은 일반 자바스크립트와는 다르다.

`{/\* 주석처리 할 내용 \*/}` 처럼 작성한다.

<br>

- `//`, `/\* \*/`을 사용하게 되면 페이지에 고스란히 나타난다.

3장

3. 컴포넌트

> ### 컴포넌트 선언방식

1. 함수 컴포넌트 (rsc 명령어 사용)

\*\*장점\*\*

`1. 클래스형 컴포넌트보다 선언하기 쉬움`

`2. 메모리 자원을 덜 사용한다.`

`3. 프로젝트를 완성하여 빌드 후 배포할 때 결과물의 파일 크기가 더 작다. (크게 중요하지는 않음.)`

\*\*단점\*\*

`1. state와 라이프사이클 API의 사용이 불가능. `

` - (Hooks 기능 도입후 해결. 조금 다른 방식으로 비슷한 작업을 할 수 있게 되었음.)`

<br>

`공식 메뉴얼에서는 함수 컴포넌트와 Hooks를 사용하도록 권장하고 있음.`

`하지만, 클래스형 컴포넌트의 기능은 알아두어야 함.`

<br>

2. 클래스형 컴포넌트 (rcc명령어 사용)

\*\*장점\*\*

`state 기능과 라이프사이클 기능 사용 가능, 임의 메서드를 정의할 수 있음.`

`render 함수가 꼭 있어야 하며, 그 안에서 보여 주어야할 JSX를 반환해야함.`

함수 컴포넌트

```

import React from 'react';

const ex = () => {

return (

<div>

</div>

);

};

export default ex;

```

`function 키워드 대신에 () => {}(화살표 함수)를 사용하여 표현하였다.`

`추후 자바스크립트 정리할 때 다시 언급할 예정.`

클래스형 컴포넌트

```

import React, { Component } from 'react';

class ex extends Component {

render() { /\* <- 필수 \*/

return (

<div>

</div>

);

}

}

export default ex;

```

코드를 작성할 때 보면 export와 import를 볼 수 있다.

이것들의 의미는 처음에 봤을 때 잘 알지 못해 '뭐지?'라는 생각이 들었었다.

따라서, 이것 또한 정리를 할 예정이다.

설명하기에 앞서 예시의 코드를 먼저 작성하도록 하자.

```

import exportEx from './exportEx'; /\* exportEx를 불러옴.\*/

const importEx = () => {

return <exportEx />;

};

export default importEx;

/\* 다른 파일에서 이 파일을 import할 때선언한 importEx 클래스를 불러오도록 함. \*/

```

>

`export` - 다른 파일에서 exportEx라는 파일을 import 할 때, 위에서 선언한 exportEx 클래스를 불러오도록 설정한다.

`import` - exportEx의 컴포넌트를 불러온다.

---

<br>

> ### props (properties)

컴포넌트 속성을 설정할 때 사용하는 요소

`값은 해당 컴포넌트를 불러와 사용하는 부모 컴포넌트에서 설정할 수 있음.`

\*\*1. JSX 내부에서 렌더링.\*\*

```

const exportEx = props => {

return (

<div> 안녕하세요 ! 제 이름은 {props.name} 입니다.</div>

/\* { }로 감싸주면 된다. \*/

);

};

export default exportEx;

```

이처럼 props를 렌더링을 할 때는 JSX 내부에서 `{ }` 기호로 감싸주면 된다.

<br>

\*\*2. 컴포넌트를 사용할 때 props 값 지정.\*\*

prop 값을 지정하려면

```

import exportEx from './exportEx'; /\* exportEx를 불러옴.\*/

const importEx = () => {

return <exportEx name="React" />;

/\* 이처럼 name="React"를 입력하면 된다.\*/

};

export default importEx;

```

`return <exportEx name= "React" />;`처럼 입력하면 된다.

<br>

\*\*3. props 기본값 설정 : defaultProps\*\*

```

const exportEx = props => {

return (

<div> 안녕하세요 ! 제 이름은 {props.name} 입니다.

exportEx.defalutProps ={

name: '기본 이름'

</div>

}; /\* <- 이처럼 지어주면 된다. \*/

);

};

export default exportEx;

```

위처럼 값을 따로 지정하지 않았을 때 보여줄 기본값을 설정하는 `defaultProps`이다.

<br>

\*\*4. 태그 사이의 내용을 보여 주는 children\*\*

리액트 컴포넌트를 사용할 때 컴포넌트 태그 사이의 내용을 보여주는 props가 있는데,

그것은 `children`이다.

```

import exportEx from './exportEx'; /\* exportEx를 불러옴.\*/

const importEx = () => {

return <exportEx>리액트</exportEx>;

};

export default importEx;

```

위 코드에서 `exportEx`태그 사이에 작성한 문자열을 `exportEx.js` 내부에서 보여주려면

props.children 값을 보여 주어야한다.

```

const exportEx = props => {

return (

<div> 안녕하세요 ! 제 이름은 {props.name} 입니다.

children 값은 {props.children}

입니다.

</div>

);

};

exportEx.defalutProps ={

name: '기본 이름'

};

export default exportEx;

```

위 코드 처럼 `{props.children}` 으로 표현을 하면 태그 사이에 있던 `리액트`가 출력되게 된다.

따라서 위 코드 결과는 아래와 같다.

>`안녕하세요 ! 제 이름은 기본 이름입니다.`

`children 값은 리액트입니다. `

\*\*5. 비구조화 할당 문법을 통해 props 내부 값 추출하기\*\*

이때까지의 코드는 `props.` 이라는 키워드를 앞에 붙였으나 더욱 편하게 하는 방법이 있다.

그것은 바로 `ES6의 비구조화 할당 문법`이라는 것이다.

이것을 사용하면 내부 값을 바로 추출할 수 있다.

위에 적었던 코드를 약간만 수정하면 된다.

```

const exportEx = props => {

const { name, children } = props; /\* <- 추가된 부분 \*/

return (

<div> 안녕하세요 ! 제 이름은 {name} 입니다.

children 값은 {children}

입니다.

</div>

);

};

exportEx.defalutProps ={

name: '기본 이름'

};

export default exportEx;

```

이처럼 객체에서 값을 추출하는 문법을 `비구조화 할당(구조 분해 문법)`이라고 부른다.

함수의 파라미터 부분에서도 사용할 수 있다.

`(객체라면 그 값을 바로 비구조화해서 사용하는 것이다.)`

조금 더 편하게 할 수 있는 방법이 있다.

```

const exportEx = ({ name, children }) => { /\* <- 수정된 부분 \*/

return (

<div> 안녕하세요 ! 제 이름은 {name} 입니다.

children 값은 {children}

입니다.

</div>

);

};

exportEx.defalutProps ={

name: '기본 이름'

};

export default exportEx;

```

방금 적었던 코드를 이렇게 수정하면 더욱 더 편하게 할 수 있다.

\*\*6. propTyes를 통한 props 검증\*\*

> `컴포넌트의 필수 props를 지정하거나 props의 타입을 지정할 때 사용한다.`

사용하려면 코드 상단에 `import`구문을 사용하여 불러와야한다.

```

import PropTypes from 'prop-types'; /\* <- 추가된 부분 \*/

const exportEx = ({ name, children }) => { /\* <- 수정된 부분 \*/

return (

<div> 안녕하세요 ! 제 이름은 {name} 입니다.

children 값은 {children}

입니다.

</div>

);

};

exportEx.defalutProps ={

name: '기본 이름'

};

exportEx.propTypes = {

name: PropTypes.string

}; /\* <- 추가된 부분 \*/

export default exportEx;

```

이렇게 설정을 하면 `name 값은 무조건 문자열 형태로 전달해야 된다`라는 것을 의미한다.

이후에

```

import exportEx from './exportEx'; /\* exportEx를 불러옴.\*/

const importEx = () => {

return <exportEx name="React">리액트</exportEx>;

};

export default importEx;

```

위 코드처럼 설정해야 Console에서 경고메세지를 출력하지 않는다.

`문자열 형태로 전달해야함.`

propTypes를 지정하지 않았을 때 경고 메시지를 띄워주려면

`propTypes를 지정할 때 뒤에 isRequired를 붙여 주면 된다.`

```

import PropTypes from 'prop-types';

const exportEx = ({ name, favoriteNumber, children }) => { /\* <- 수정된 부분 \*/

return (

<div> 안녕하세요 ! 제 이름은 {name} 입니다.

children 값은 {children}

입니다.

<br />

제가 좋아하는 숫자는 {favoriteNumber} 입니다. /\* <- 추가된 부분 \*/

</div>

);

};

exportEx.defalutProps ={

name: '기본 이름'

};

exportEx.propTypes = {

name: PropTypes.string

favoriteNumber: PropTypes.number.isRequired /\* <- 추가된 부분 \*/

};

export default exportEx;

```

추가된 부분 처럼 추가한 후 아래 코드와 작성하면 된다.

```

import exportEx from './exportEx'; /\* exportEx를 불러옴.\*/

const importEx = () => {

return <exportEx name="React" favoriteNumber={1}>리액트</exportEx>;

};

export default importEx;

```

더 많은 PropTypes 종류가 많지만, 따로 적지는 않으려고 한다.

`자세한 정보는 https://github.com/facebook/prop-types 에서 확인할 수 있다.`

---

7. 클래스형 컴포넌트에서 props 사용하기.

> 클래스형 컴포넌트에서 props를 사용할 때는 `render 함수에서 this.props를 조회`하면 된다.

`defaultProps와 propTypes는 똑같은 방식으로 설정할 수 있다.`

```

import { Component } from 'react';

import PropTypes from 'prop-types';

class exportEx extends Component {

render() {

const {name, favoriteNumber, children } = this.props; // 비구조화 할당

return (

<div> 안녕하세요 ! 제 이름은 {name} 입니다.

children 값은 {children}

입니다.

<br />

제가 좋아하는 숫자는 {favoriteNumber} 입니다.

</div>

);

}

}

exportEx.defalutProps ={

name: '기본 이름'

};

exportEx.propTypes = {

name: PropTypes.string

favoriteNumber: PropTypes.number.isRequired /\* <- 추가된 부분 \*/

};

export default exportEx;

```

위 코드처럼 클래스 외부에서 지정하는 방식과

```

import { Component } from 'react';

import PropTypes from 'prop-types';

class exportEx extends Component {

static defaultProps = {

name: '기본 이름'

};

static propTypes = {

name: this.propTypes.string,

favoriteNumber: PropTypes.number.isRequired

}

render() {

const {name, favoriteNumber, children } = this.props; // 비구조화 할당

return (

<div> 안녕하세요 ! 제 이름은 {name} 입니다.

children 값은 {children}

입니다.

<br />

제가 좋아하는 숫자는 {favoriteNumber} 입니다.

</div>

);

}

}

export default exportEx;

```

`defaultProps와 propTypes`를 클래스 내부에 지정하는 방법도 있다.

> 해당 컴포넌트에 어떤 props가 필요한지 쉽게 알 수 있으 개발 능률이 좋아지므로 `defaultProps와 propTypes`를 사용하는 것이 좋다.

---

> ### 4. state

` 리액트에서 state는 컴포넌트 내부에서 바뀔 수 있는 값을 의미`

`1. 클래스형 컴포넌트가 지니고 있는 state`

`2. 함수 컴포넌트에서 useState라는 함수를 통해사용하는 state` 가 있다.

>> ` props는 컴포넌트가 사용되는 과정에서 부모 컴포넌트가 설정하는 값`

` 컴포넌트 자신은 해당 props를 읽기 전용으로만 사용.`

` props를 바꾸려면 부모 컴포넌트에서 바꿔줘야 한다.`

### 3.4.1 클래스형 컴포넌트의 state

```

import { Component } from 'react';

class Counter extends Component {

constructor(props) {

super(props);

//state의 초깃값 설정하기

this.state = {

number: 0

};

}

render() {

const { number } = this.state; // state를 조회할 때는 this.state로 조회한다.

return (

<div>

<h1>{number}</h1>

<button onClick={() => { // onclick을 통해 버튼이 클릭되었을 때 호출할 함수를 지정.

this.setState({ number: number +1 }); // this.setState를 사용하여 state에 새로운 값을 넣을 수 있다.

}}

>

+1

</button>

</div>

);

}

}

export default Counter;

```

이 코드에서 각 어떤 역할을 하는지 알아보자.

```

constructor(props) {

super(props); // <- 반드시 호출해야함.

//state의 초깃값 설정하기

this.state = {

number: 0

};

}

```

>이 부분은 `컴포넌트의 생성자 메서드`이다.

1. 클래스형 컴포넌트에서 `constructor`를 작성할 때는 반드시 `super(props)`를 호출해주어야 한다.

2. 이 함수가 호출되면 현재 클래스형 컴포넌트가 상속받고 있는

`리액트의 Component 클래스가 지닌 생성자 함수를 호출`해준다.

3. `this.state`값에 초깃값을 설정해주었으며, 컴포넌트의 state는 `객체 형식`이어야한다.

```

render() {

const { number } = this.state; // state를 조회할 때는 this.state로 조회한다.

return (

<div>

<h1>{number}</h1>

<button onClick={() => { // onclick을 통해 버튼이 클릭되었을 때 호출할 함수를 지정.

this.setState({ number: number +1 }); // this.setState를 사용하여 state에 새로운 값을 넣을 수 있다.

}}

>

+1

</button>

</div>

);

}

```

> 1. render 함수에서 state를 조회할 때는 `this.state`를 조회하면 된다.

2. button 안에 onClick이라는 값을 props로 넣어 주었는데 이는 버튼이 클릭될 때 호출시킬 함수를 설정할 수 있게 해준다. `(<- 이벤트를 설정한다는 뜻이다.)`

2-1. 이벤트를 설정할 함수를 넣어줄 때는 화살표 함수 문법을 사용해야한다.

#### 3.4.1.1. state 객체 안에 여러 값이 있을 때

state 객체 안에는 아래의 코드처럼 여러 값이 있을 수 있다.

```

import { Component } from 'react';

class Counter extends Component {

constructor(props) {

super(props);

//state의 초깃값 설정하기

this.state = {

number: 0,

fixedNumber: 0 // <- 추가된 부분

};

}

render() {

const { number, fixedNumber } = this.state; // state를 조회할 때는 this.state로 조회한다.

return (

<div>

<h1>{number}</h1>

<h2>바뀌지 않는 값: {fixedNumber}</h2> // <- 추가된 부분

<button onClick={() => { // onclick을 통해 버튼이 클릭되었을 때 호출할 함수를 지정.

this.setState({ number: number +1 }); // this.setState를 사용하여 state에 새로운 값을 넣을 수 있다.

}}

>

+1

</button>

</div>

);

}

}

export default Counter;

```

이처럼 `fixedNumber`를 추가했지만 `this.setState 함수`인자로 전달되는 개체 내부에는 `fixedNumber`를 넣어 주지는 않았다.

따라서, `this.setState 함수`는 `number` 값만 바꾸어 준다.

#### 3.4.1.2 state를 constructor에서 꺼내기

```

import { Component } from 'react';

class Counter extends Component {

state = {

number: 0,

fixedNumber: 0

};

render() {

const { number, fixedNumber } = this.state; // state를 조회할 때는 this.state로 조회한다.

return (

<div>

<h1>{number}</h1>

<h2>바뀌지 않는 값: {fixedNumber}</h2>

<button onClick={() => { // onclick을 통해 버튼이 클릭되었을 때 호출할 함수를 지정.

this.setState({ number: number +1 }); // this.setState를 사용하여 state에 새로운 값을 넣을 수 있다.

}}

>

+1

</button>

</div>

);

}

}

export default Counter;

```

이렇게 하면 `constructor 메서드`를 선언하지 않고도 state 초깃값을 설정할 수 있다.

#### 3.4.1.3 this.setState에 객체 대신 함수 인자 전달하기

> `this.setState`를 사용하여 state 값을 업데이트할 때는 상태가 비동기적으로 업데이트 된다.

예를 들어 onClick에 설정한 함수 내부에서 `this.setState`를 두 번 호출해보게 될 경우

```

<button onClick={() => {

this.setState({ number: number +1 });

this.setState({ number: this.state.number + 1});

}}

```

`this.setState`를 두 번 사용하는 것임에도 버튼을 클릭할 때 1 만 더해진다.

그 이유는 `this.setState`를 사용한다고 해서 `state 값`이 바뀌지 않기 때문이다.

바뀌게 하려면 어떻게 해야할까?

그 방법은 `this.setState`를 사용할 때 \*\*객체 대신에 함수를 인자로 넣어 주는 것\*\*이다.

`(코드는 아래와 같다.)`

```

this.setState((prevState, props) => {

return {

// 업데이트 하고 싶은 내용

}

})

```

>여기서 `prevState`는 기존 상태, `props`는 현재 지니고 있는 props를 가리킨다.

만약 업데이트 하는 과정에서 props가 필요하지 않다면 생략을 해도 된다.

기존의 코드를 수정하면

```

<button onClick={() => {

this.setState(prevState => {

return { prevState.number+1

};

});

// 위 아래 두 개의 코드는 같은 기능을 한다.

this.setState(prevState => ({

number: prevState.number +1 // 해당 코드는 함수에서 바로 객체를 반환한다는 의미이다.

}));

}}

>

+1

</button>

```

#### 3.4.1.4 this.setState가 끝난 후 특정 작업 실행하기

>setState를 사용하여 값을 업데이트하고 난 다음에 특정 작업을 하고 싶을 때는

`setState의 두 번째 파라미터로 콜백함수를 등록하여 작업을 처리하면 된다.`

onClick 함수를 다시 한번 수정해보자.

```

<button onClick={() => {

this.setState(

{

number: number +1

},

() => {

console.log('방금 setState가 호출되었다.');

console.log(this.state);

}

);

}}

>

+1

</button>

```

콘솔 탭을 확인하면 `console.log`로 나타낸 것이 출력되는 것을 볼 수 있다.

### 3.4.2 함수 컴포넌트에서 useState 사용하기.

> 해당 부분은 `Hooks`라는 것을 사용한다. 추후에 나올 부분에서 제대로 공부하자.

#### 3.4.2.1 배열 비구조화 할당

> `배열 비구조화 할당 - 배열 안에 들어 있는 값을 쉽게 추출할 수 있도록 해주는 문법`

>```

const array = [1, 2];

const one = array[0];

const two = array[1];

```

```

const array =[1, 2];

const [one,two] = array;

```

이 두가지 코드는 동일한 것으로 아래 코드가 더욱 깔끔한 것을 볼 수 있다.

#### 3.4.2.2 useState 사용하기

```

import { useState } from 'react';

const Say = () => {

const [message, setMessage] = useState('');

const onClickEnter = () => setMessage('안녕하세요!');

const onClickLeave = () => setMessage('안녕히 가세요!');

return (

<div>

<button onClick = {onClickEnter}>입장</button>

<button onClick = {onClickLeave}>퇴장</button>

<h1>{message}</h1>

</div>

);

};

export default Say;

```

이 코드를 설명해보자.

useState 함수 인자에는 상태의 초기값을 넣어준다.

`(useState에서는 반드시 객체가 아니여도 상관없다. 값의 형태는 자유이다.`

`숫자일 수도, 문자열일 수도, 객체일 수도, 배열일 수도 있다.)`

함수를 호출하면 배열이 반환된다.

배열의 첫 번째 원소는 `현재 상태`

두 번째 원소는 `상태를 바꾸어 주는 함수`이다.

이 함수를 `세터 함수`라고 부른다.

(배열 비구조화 할당을 통해 이름을 자유롭게 정해 줄 수 있다.)

#### 3.4.2.3 한 컴포넌트에서 useState 여러 번 사용하기

> 한 컴포넌트에서 useState를 여러번 사용해도 상관없다.

아래 코드를 보면 useState를 두 번 사용한 것을 볼 수 있다.

```

import { useState } from 'react';

const Say = () => {

const [message, setMessage] = useState('');

const onClickEnter = () => setMessage('안녕하세요!');

const onClickLeave = () => setMessage('안녕히 가세요!');

const [color, setColor] = useState('black');

return (

<div>

<button onClick = {onClickEnter}>입장</button>

<button onClick = {onClickLeave}>퇴장</button>

<h1 style = {{ color }}>{message}</h1>

<button style={{ color: 'red' }} onCluck={() => setColor('red')}>

빨간색

</button>

<button style={{ color: 'green' }} onCluck={() => setColor('greed')}>

초록색

</button>

<button style={{ color: 'blue' }} onCluck={() => setColor('blue')}>

파란색

</button>

</div>

);

};

export default Say;

```

---

### 3.5 state를 사용할 때 주의 사항.

>state값을 바꾸어야 할 때는 setState 혹은 useState를 통해 전달받은 세터 함수를 사용해야한다.

## 4.1 리액트의 이벤트 시스템

> ### 4.1.1 주의사항

1. 이벤트 이름은 `카멜 표기법`으로 작성할 것.

`onclick -> onClick`

2. 이벤트에 실행할 자바스크립트 코드를 전달하는 것이 아니라, `함수 형태의 값을 전달`해야 함.

3. DOM 요소에만 이벤트를 설정할 수 있다.

`직접 만든 컴포넌트에는 이벤트를 자체적으로 설정할 수 없음.`

> ### 4.1.2 이벤트 종류

`Clipboard` &nbsp&nbsp&nbsp&nbsp&nbsp `Touch`

`Composition`&nbsp&nbsp&nbsp&nbsp` UI `

`Keyboard`&nbsp&nbsp&nbsp&nbsp&nbsp&nbsp&nbsp&nbsp`Wheel`

`Focus` &nbsp&nbsp&nbsp&nbsp&nbsp&nbsp&nbsp&nbsp&nbsp&nbsp&nbsp&nbsp`Media`

`Form` &nbsp&nbsp&nbsp&nbsp&nbsp&nbsp&nbsp&nbsp&nbsp&nbsp&nbsp&nbsp&nbsp`Image`

`Mouse`&nbsp&nbsp&nbsp&nbsp&nbsp&nbsp&nbsp&nbsp&nbsp&nbsp&nbsp&nbsp`Animation`

`Selection` &nbsp&nbsp&nbsp&nbsp&nbsp&nbsp`Transition`

<br>

더 많은 이벤트는 `리액트 메뉴얼` https://facebook.github.io/react/docs/events.html 에서 볼 수 있다고 한다

## 4.2 이벤트 핸들링 익히기

### 4.2.1 컴포넌트 생성 및 불러오기

#### 4.2.1.1 컴포넌트 생성.

>```

import { Component } from 'react';

class EventPractice extends Component {

render() {

return (

<div>

<h1>이벤트 연습</h1>

</div>

);

}

}

export default EventPractice;

```

#### 4.2.1.2 렌더링

>```

import EventPractice from './EventPractice';

const App = () => {

return <EventPractice />

};

export default App;

```

### 4.2.2 onChange 이벤트 핸들링하기

#### 4.2.2.1 onChange 이벤트 설정.

>```

import { Component } from 'react';

class EventPractice extends Component {

render() {

return (

<div>

<h1>이벤트 연습</h1>

<input

type="text"

name="message"

placeholder="아무거나 입력해 보세요"

onChange={

(e) => {

console.log(e.target.value);

}

}

/>

</div>

);

}

}

export default EventPractice;

```

렌더링을 하게 되면 `e.target.value`을 통해 값이 바뀔 때마다 바뀌는 값이 콘솔에 기록이 된다.

#### 4.2.2.2 버튼을 누를 때 comment 값을 공백으로 설정

>

입력한 값이 `state`에 잘 들어갔는지와 `input`에서 그 값을 제대로 반영하는지 검증을 해보자.

```

import { Component } from 'react';

class EventPractice extends Component {

state = {

message: ''

}

render() {

return (

<div>

<h1>이벤트 연습</h1>

<input

type="text"

name="message"

placeholder="아무거나 입력해 보세요"

value={this.state.message}

onChange={

(e) => {

this.setState({

message: e.target.value

})

}

}

/>

<button onClick={

() => {

alert(this.state.message);

this.setState({

message: ''

});

}

}>확인</button>

</div>

);

}

}

export default EventPractice;

```

위 코드에서는 클릭 이벤트가 발생하면 comment 값을 메세지로 띄운 후 공백으로 설정하도록 되어 있다.<br>

`(state - 컴포넌트 내부에서 바뀔 수 있는 값.)`

### 4.2.3 임의 메서드 만들기

> `이벤트에 실행할 자바스크립트 코드를 전달하는 것이 아니라, 함수 형태의 값을 전달한다.`라고하였으나,

\*\*함수를 미리 준비하여 전달\*\*하는 방법도 있다.

성능의 차이는 없지만 가독성은 훨씬 좋다.(상황에 따라 렌더링 메서드 내부에서 함수를 만드는 것이 더 편할 때도 있다.)

<br>

```

import { Component } from 'react';

class EventPractice extends Component {

state = {

message: ''

}

constructor(props){

super(props);

this.handleChange = this.handleChange.bind(this);

this.handleClick = this.handleClick.bind(this);

}

handleChange(e) {

this.setState({

message: e.target.value

});

}

handleClick(){

alert(this.state.message);

this.setState({

message:''

});

}

render() {

return (

<div>

<h1>이벤트 연습</h1>

<input

type="text"

name="message"

placeholder="아무거나 입력해 보세요"

value={this.state.message}

onChange={this.handleChange}

/>

<button onClick={this.handleClick}>확인</button>

</div>

);

}

}

export default EventPractice;

```

이제 코드를 해석해보자.

함수가 호출될 때는 `this`는 호출부에 따라 결정된다.

이에 따라 클래스의 임의 메서드가 특정 HTML 요소의 이벤트로 등록되는 과정에서 메서드와 this의 관계가 끊어져 버린다.

이결 해결하기 위해서는 `임의 메서드가 이벤트로 등록되어도 this를 컴포넌트 자신으로 가리키는 메서드와 바인딩하는 작업`이 필요하다. <br>

그래서 `constructor` 함수에서 보면

```

constructor(props){

super(props);

this.handleChange = this.handleChange.bind(this);

this.handleClick = this.handleClick.bind(this);

}

```

`bind(this)`로 함수를 바인딩하는 작업이 이루어지는 것을 볼 수 있다.

#### 4.2.3.2 Property Initializer Syntax를 사용한 메서드 작성.

>메서드 바인딩은 생성자 메서드에서 하는 것이 정석이지만 해당 작업이 불편하다고 느낄 경우 간단하게 할 수 있는 방법이 있다. 그것은 `바벨의 transform-class-properties 문법`을 사용하여 화살표 함수 형태로 메서드를 정의하는 것이다.

해당 문법을 사용할 경우

```

import { Component } from 'react';

class EventPractice extends Component {

state = {

message: ''

}

handleChange = (e) => {

this.setState({

message: e.target.value

});

}

handleClick = () =>{

alert(this.state.message);

this.setState({

message:''

});

}

render() {

return (

<div>

<h1>이벤트 연습</h1>

<input

type="text"

name="message"

placeholder="아무거나 입력해 보세요"

value={this.state.message}

onChange={this.handleChange}

/>

<button onClick={this.handleClick}>확인</button>

</div>

);

}

}

export default EventPractice;

```

이제 코드를 해석해보자.

```

constructor(props){

super(props);

this.handleChange = this.handleChange.bind(this);

this.handleClick = this.handleClick.bind(this);

}

```

`handleChange(e)` 가 ` handleChange = (e) => {...}` 로,

`handleClick()` 가 `handleClick = () =>{...}`로 바뀌었으며,

`constructor` 함수가 없어졌다. 하지만, 결과는 똑같이 실행되는 것을 볼 수 있다.

### 4.2.4 input 여러 개 다루기

>`input`이 여러 개 일 경우에는 메서드를 여러 개 만들 수 있지만, 더 쉽게 처리하는 방법이 있다.

그것은 `event 객체`를 활용하는 것인데 `e.target.name`값을 사용하면 된다.

```

import { Component } from 'react';

class EventPractice extends Component {

state = {

username: '',

message: ''

}

handleChange = (e) => {

this.setState({

[e.target.name]: e.target.value

});

}

handleClick = () =>{

alert(this.state.username + ': ' + this.state.message);

this.setState({

username: '',

message: ''

});

}

render() {

return (

<div>

<h1>이벤트 연습</h1>

<input

type="text"

name="username"

placeholder="사용자명"

value={this.state.username}

onChange={this.handleChange}

/>

<input

type="text"

name="message"

placeholder="아무거나 입력해 보세요"

value={this.state.message}

onChange={this.handleChange}

/>

<button onClick={this.handleClick}>확인</button>

</div>

);

}

}

export default EventPractice;

```

>해당 코드를 해석하면 기존의 코드에서 `name`값으로 `username`을 input에 렌더링하였으며, state쪽에도 `username` 값을 추가하였다.

또, 그에 맞게 handleChange도 변경하였다.

위 코드의 핵심은

```

handleChange = (e) => {

this.setState({

[e.target.name]: e.target.value

});

}

```

이 부분으로

객체 안에서 key를 [ ]로 감싸면 그 안에 넣은 레퍼런스가 가리키는 실제 값이 key 값으로 사용된다.

조금 더 쉽게 설명하자면

```

const name = 'varaianKey';

const object = {

[name]: 'value'

};

```

의 결과는 다음과 같다.

```

{

'variantKey': 'value'

}

```

name을 [ ]로 감싸면 name의 실제 값인 `varaianKey`가 key 값으로 사용되는 것을 볼 수 있다.

### 4.2.5 onKeyPress 이벤트 핸들링

> `Enter`를 눌렀을 때 `handleClick` 메서드를 호출하도록 코드를 작성해보자.

```

import { Component } from 'react';

class EventPractice extends Component {

state = {

username: '',

message: ''

}

handleChange = (e) => {

this.setState({

[e.target.name]: e.target.value

});

}

handleClick = () =>{

alert(this.state.username + ': ' + this.state.message);

this.setState({

username: '',

message: ''

});

}

// ------------

handleKeyPress = (e) => {

if(e.key === 'Enter'){

this.handleClick();

}

}

// ----------------

// <- 추가된 부분

render() {

return (

<div>

<h1>이벤트 연습</h1>

<input

type="text"

name="username"

placeholder="사용자명"

value={this.state.username}

onChange={this.handleChange}

/>

<input

type="text"

name="message"

placeholder="아무거나 입력해 보세요"

value={this.state.message}

onChange={this.handleChange}

onKeyPress={this.handleKeyPress} // <- 추가된 부분

/>

<button onClick={this.handleClick}>확인</button>

</div>

);

}

}

export default EventPractice;

```

기존의 코드에서 표시된 부분을 추가했다.

![](https://images.velog.io/images/hyuri/post/625e3c3d-7c7e-4fc5-862c-20fc6632e8a6/image.png)

엔터를 눌렀을 때 위 화면처럼 출력되는 것을 볼 수 있다.

---

## 4.3 함수 컴포넌트로 구현해 보기

>기존의 컴포넌트를 함수 컴포넌트로 작성해보자.

```

import { useState } from "react";

const EventPractice = () => {

const [username, setUsername] = useState('');

const [message, setMessage] = useState('');

const onChangeUsername = e => setUsername(e.target.value);

const onChangeMessage = e => setMessage(e.target.value);

const onClick = () => {

alert(username + ': ' +message);

setUsername('');

setMessage('');

};

const onKeyPress = e => {

if (e.key === 'Enter') {

onClick();

}

};

return (

<div>

<h1>이벤트 연습</h1>

<input

type="text"

name="username"

placeholder="사용자명"

value={username}

onChange={onChangeUsername}

/>

<input

type="text"

name="message"

placeholder="아무거나 입력해 보세요"

value={message}

onChange={onChangeMessage}

onKeyPress={onKeyPress}

/>

<button onClick={onClick}>확인</button>

</div>

);

};

export default EventPractice;

```

> 해당 코드에서는 `e.target.name`을 활용하지 않고 `onChange 관련 함수` 두 개를 따로 만들어주었다.

input이 두 개밖에 없다면 괜찮지만 input의 갯 수가 많아질 것 같으면 `e.target.name`을 활용하는 것이 좋다.

```

import { useState } from "react";

const EventPractice = () => {

const [form, setForm] = useState({

username: '',

message: ''

});

const { username, message} = form;

const onChange = e => {

const nextForm = {

...form, // 기존의 form 내용을 이 자리에 복사한 뒤

[e.target.name]: e.target.value // 원하는 값을 덮어 씌우기

};

setForm(nextForm);

}

const onClick = () => {

alert(username + ': ' +message);

setForm({

Username: '',

Message: ''

});

};

const onKeyPress = e => {

if (e.key === 'Enter') {

onClick();

}

};

return (

<div>

<h1>이벤트 연습</h1>

<input

type="text"

name="username"

placeholder="사용자명"

value={username}

onChange={onChange}

/>

<input

type="text"

name="message"

placeholder="아무거나 입력해 보세요"

value={message}

onChange={onChange}

onKeyPress={onKeyPress}

/>

<button onClick={onClick}>확인</button>

</div>

);

};

export default EventPractice;

```

해당 코드에서

```

const EventPractice = () => {

const [form, setForm] = useState({

username: '',

message: ''

});

const { username, message} = form;

const onChange = e => {

const nextForm = {

...form,

[e.target.name]: e.target.value

};

setForm(nextForm);

}

```

부분을 통해 4.2.4에서 했던 input 여러 개를 다루는 법을 사용하였다.

---

여기서 리액트의 장점이 있는데 그것은

`자바스크립트에 익숙하다면 쉽게 활용할 수 있다는 것`이다.

이 말은 즉슨, 기`존의 HTML DOM Event를 알고 있다면 리액트의 컴포넌트 이벤트도 쉽게 다룰 수 있다는 의미`가 된다.

일반 HTML에서 `DOM(Document Object Model) 요소`에 이름을 달 떄는 `id`를 사용한다.

```

<div id="my-id"></div>

```

특정 DOM 요소에 id를 달면 CSS에서 특정 id에 특정 스타일을 적용하거나, 자바스크립트에서 해당 id를 가진 요소를 찾아서 작업할 수 있다.

HTML에서 id를 사용하여 DOM에 이름을 다는 것 처럼

제목에서 보면 `ref(reference의 줄임말)`은 리액트 내부에서 DOM에 이름을 다는 방법이다.

>`id`를 사용할 수 있지만, 컴포넌트를 만들 때마다 id 뒷 부분에 추가 텍스트를 붙여서 중복 id가 발생하는 것을 방지해야만 한다.

이에 반해 `ref`는 컴포넌트 내부에서만 작동하므로 중복 id 사용을 방지할 수 있다.

---

## 5.1 ref는 어떤 상황에서 사용해야 할까?

> `DOM을 꼭 ! 직접적으로 건드려야 할 때 사용한다.`

자바스크립트 및 jQuery로 만든 웹사이트에서 input을 검증할 때는 특정 id를 가진 input에 클래스를 설정해서 검증을 한다.

하지만, 리액트에서는 굳이 DOM에 접근하지 않아도 state로 구현할 수 있다.

(지금부터는 예제 코드를 작성하면서 해야 이해가 잘 되므로 코드를 작성하면서 작성할 예정이다.)

`ValidationSample.css`와 `ValidationSample.js`를 만들어 보자

<br>

`ValidationSample.css`

```

.success {

background-color: lightgreen;

}

.failure {

background-color: lightcoral;

}

```

<br>

`ValidationSample.js`

```

import { Component } from 'react';

import 'ValidationSample.css';

class ValidationSample extends Component {

state = {

password: '',

clicked: false,

validated: false

}

handleChange = (e) => {

this.setState({

password: e.target.value

});

}

handleButtonClick = () => {

this.setState({

clicked: true,

validated: this.state.password === '0000'

})

}

render() {

return (

<div>

<input

type="password"

value={this.state.password}

onChange={this.handleChange}

className={this.state.clicked ? (this.state.validated ? 'success' : 'failure') : ''}

/>

<button onClick={this.handleButtonClick}>검증하기</button>

</div>

);

}

}

export default ValidationSample;

```

이제 코드를 해석해보자.

>

```

<input

type="password"

value={this.state.password}

onChange={this.handleChange}

className={this.state.clicked ? (this.state.validated ? 'success' : 'failure') : ''}

/>

```

여기서, `onChange`이벤트가 발생하면 `handleChance`를 호출한다.

또한, `className`값은 버튼을 누르기 전에는 비어있는 문자열을 전달한다.

```

handleChange = (e) => {

this.setState({

password: e.target.value

});

}

```

그 이후 호출하면 `state`의 `password` 값이 업데이트 되게 했다.

>

```

<button onClick={this.handleButtonClick}>검증하기</button>

```

버튼에서는 `onClick` 이벤트가 발생하면 `handleButtonClick`을 호출한 후,

```

handleButtonClick = () => {

this.setState({

clicked: true,

validated: this.state.password === '0000'

})

}

```

`click`값을 `true`로 설정했고, `validated` 값을 검증 결과로 설정했다.

버튼을 누른 후에는 검증 결과에 따라 `success값`과 `failure값`을 설정한다.

또한, 해당 값에 따라 `ValidationSample.css`을 통해 input의 색깔이 달라진다.

![](https://images.velog.io/images/hyuri/post/58806676-c398-4d7f-b7fa-f601406d4c69/image.png)

`아무것도 입력하지 않고 버튼 클릭`

![](https://images.velog.io/images/hyuri/post/101c9ad2-4abc-430c-bef4-399818772650/image.png)

`0000을 입력한 후 버튼 클릭`

### 5.1.2 App 컴포넌트에서 예제 컴포넌트 렌더링

```

import { Component } from 'react';

import ValidationSample from './ValidationSample';

class App extends Component {

render() {

return (

<ValidationSample />

);

}

}

export default App;

```

>

`App` 컴포넌트에서 `ValidationSample` 컴포넌트를 불러와 렌더링 하였다.

(ref를 사용할 것이기 때문에 `함수`컴포넌트가 아닌 `클래스형` 컴포넌트로 작성하였다.)

### 5.1.3 DOM을 꼭 사용해야 하는 상황

> - 특정 input에 포커스 주기

- 스크롤 박스 조작하기

- Canvas 요소에 그림 그리기 등<br>

이때는 어쩔 수 없이 DOM에 직접적으로 접근해야 하는 상황이 발생한다.

이를 위해서 `ref`를 사용한다.

---

## 5.2 ref 사용

### 5.2.1 콜백함수를 통한 ref 설정

> `ref`를 만드는 가장 기본적인 방법은 `콜백 함수`를 사용하는 것이다.

`ref`를 달고자 하는 요소에 `ref`를 콜백 함수를 `props`로 전달해 주면 된다.<br>

이렇게 할 경우 콜백 함수는 `ref` 값을 파라미터로 전달 받고, 함수 내부에서 파라미터로 받은 ref를 컴포넌트의 멤버 변수로 설정해준다.

```

<input ref={(ref)} => {this.input=ref}} />

```

사용하기에 앞서 예시를 살펴보자.

`this.input`은 inpute 요소의 DOM을 가리킨다.

`ref`의 이름은 원하는 것으로 자유롭게 지정할 수 있으며, DOM 타입과 관계없이 `this.superman = ref` 처럼 마음대로 지정한다.

### 5.2.2 createRef를 통한 ref 설정

> `ref`를 만드는 또 다른 방법은 리액트에 내장되어 있는 `createRef`라는 함수를 사용하는 것이다.

```

import { Component } from 'react';

class RefSample extend Component {

input = React.createRef(); // 1번

handleFocus = () => {

this.input.current.focus(); // 3번

}

render() {

return (

<div>

<input ref={this.input} /> // 2번

</div>

);

}

}

export default RefSample;

```

해당 코드는 `createRef`의 사용 예시이다. 지금부터 해석을 한 번 해보자.

`createRef`를 사용하여 `ref`를 만들려면,

1. 우선 컴포넌트 내부에서 멤버 변수로 `React.createRef()`를 담아주어야한다.

2. 해당 멤버 변수를 `ref` 달고자 하는 요소에 `ref props`로 넣어주면 `ref` 설정이 완료된다.

3. 설정한 후 나중에 설정해 준 DOM에 접근하려면 `this.input.current`를 조회하면 된다.

( 콜백 함수를 사용할 때와의 차이점은 뒷 부분에 `.current`를 넣어주어야 한다는 것이다.

>버튼을 클릭 했을 때 포커스가 다시 `input` 쪽으로 자동으로 넘어가게 해보자.

#### 5.2.3.1-2 inpute에 ref 달기, 버튼 onClick 이벤트 코드 수정

기존의 ValidationSample 컴포넌트에서 ref를 달아보자

```

import { Component } from 'react';

import './ValidationSample.css';

class ValidationSample extends Component {

state = {

password: '',

clicked: false,

validated: false

}

handleChange = (e) => {

this.setState({

password: e.target.value

});

}

handleButtonClick = () => {

this.setState({

clicked: true,

validated: this.state.password === '0000'

});

this.input.focus();

}

render() {

return (

<div>

<input

ref={(ref) => this.input=ref} // <- 추가한 내용

type="password"

value={this.state.password}

onChange={this.handleChange}

className={this.state.clicked ? (this.state.validated ? 'success' : 'failure') : ''}

/>

<button onClick={this.handleButtonClick}>검증하기</button>

</div>

);

}

}

export default ValidationSample;

```

추가로 버튼에서 onClick 이벤트가 발생했을 때 input에 포커스를 주도록 코드를 수정해보자

`this.input`이 컴포넌트 내부의 `input`요소를 가리키고 있는 것을 알 수 있다.

이제 실행하고 버튼을 클릭하면 input으로 포커스가 넘어간 것을 볼 수 있다.

---

## 5.3 컴포넌트에 ref 달기

>리액트에서는 컴포넌트에도 `ref`를 달 수 있다.

이 방법은 주로 컴포넌트 내부에 있는 DOM을 컴포넌트 외부에서 사용할 때 쓴다.

방법은 DOM에 ref를 다는 방법과 같다.

### 5.3.1 사용법

```

<MyComponent

ref = {(ref) => {this.myComponent=ref}}

/>

```

이렇게 하면 `MyComponent` 내부의 메서드 및 멤버 변수에도 접근할 수 있다.

즉, 내부의 `ref`에도 접근할 수 있다.

<br>

스크롤 박스가 있는 컴포넌트와 스크롤바를 아래로 내리는 작업을 부모 컴포넌트에서 실행해보자.

### 5.3.2 컴포넌트 초기 설정

>

- `ScrollBox`라는 컴포넌트 파일을 만들기.

- JSX의 인라인 스타일링 문법으로 스크링 박스를 만들기.

- 최상위 DOM에 ref를 달기

순으로 진행해보자.

#### 5.3.2.1 컴포넌트 파일 생성

```

import { Component } from 'react';

class ScrollBox extends Component {

render() {

const style = {

border: '1px solid black',

height: '300px',

width: '300px',

overflow: 'auto',

position: 'relative'

};

const innerStyle = {

width: '100%',

height: '650px',

background: 'linear-gradient(white, black)'

}

return (

<div

style={style}

ref={(ref) => {this.box=ref}}>

<div style = {innerStyle}/>

</div>

);

}

}

export default ScrollBox;

```

코드를 해석해보자

```

<div

style={style}

ref={(ref) => {this.box=ref}}>

<div style = {innerStyle} />

</div>

```

컴포넌트에 `ref` 달아놨다.

여기에서 `style={style}`은 `render`함수 안에 있는 `style`를 불러온다.

`<div style = {innerStyle}`에서는 스크롤 박스 안에 나타낼 코드를 보이고 있다.

이후에 `App.js`를 수정하여 렌더링을 해보면

![](https://images.velog.io/images/hyuri/post/c539e18e-1de6-45ba-8458-c207c550ed8c/image.png)

이런식으로 나타나는 것을 볼 수 있다.

### 5.3.3 컴포넌트에 메서드 생성

> 이번에는 컴포넌트에 스크롤바를 맨 아래쪽으로 내리는 메서드를 만들어보자.

자바스크립트로 스크롤바를 내릴 때는 DOM 노드가 가진 다음 값들을 사용한다.

- `scrollTop`: 세로 스크롤바 위치(0~350)

- `scrollHeight`: 스크롤이 있는 박스 안의 div 높이(650)

- `clientHeight`: 스크롤이 있는 박스의 높이(300) <br>

이제 추가해보자.

```

import { Component } from 'react';

class ScrollBox extends Component {

scrollToBottom = () => {

const { scrollHeight, clientHeight } = this.box;

/\* 앞 코드에는 비구조화 할당 문법을 사용했다.

해당 코드는 다음코드는 같은 의미이다.

const scrollHeight = this.box.scrollHeight;

const clientHeight = this.box.clientHeight;

\*/

this.box.scrollTop = scrollHeight - clientHeight;

}

render() {

const style = {

border: '1px solid black',

height: '300px',

width: '300px',

overflow: 'auto',

position: 'relative'

};

const innerStyle = {

width: '100%',

height: '650px',

background: 'linear-gradient(white, black)'

}

return (

<div

style={style}

ref={(ref) => {this.box=ref}}>

<div style = {innerStyle}/>

</div>

);

}

}

export default ScrollBox;

```

이제 해석해보자.

`scrollToBottom` 메서드에서 첫 번째 줄에서는 ES6의 `비구조화 할당 문법`을 사용했다.

여기서, `scrollHeight`에서 `clientHeight` 높이를 빼면 스크롤바를 맨 아래쪽으로 내릴 수 있다.

### 5.3.4 컴포넌트에 ref 달고 내부 메서드 사용

`App` 컴포넌트에서 ScrollBox에 `ref`를 달고 버튼을 만들어 누르면, ScrollBox 컴포넌트의 `scrollToBottom` 메서드를 실행하도록 코드를 작성해보자.

```

import { Component } from 'react';

import ScrollBox from './ScrollBox';

class App extends Component {

render() {

return (

<div>

<ScrollBox ref={(ref) => this.scrollBox=ref}/>

<button onClick={() => this.scrollBox.scrollToBottom()}>

맨 밑으로

</button>

</div>

);

}

}

export default App;

```

버튼을 누르면

![](https://images.velog.io/images/hyuri/post/dc1fb018-1380-4a18-ba08-927e96bff8d0/image.png)

이처럼 맨 아래로 이동하는 것을 볼 수 있다.

<br><br>

### 주의할 점이 하나 있다.

문법상으로는 `onClick = {this.scrollBox.scrollBottom}` 같은 형식으로 작성해도 틀린 것은 아니다.

하지만, 컴포넌트가 처음 렌더링 될 때는 `this.scrollBox` 값이 `undefined`이므로 `this.scrollBox.scrollToBottom` 값을 읽어오는 과정에서 오류가 발생한다.

화살표 함수 문법을 사용하여 아예 새로운 함수를 만들고 그 내부에서 `this.scrollBox.scrollToBottom` 메서드를 실행하면,

버튼을 누를 때 `this.scrollBox.scrollToBottom` 값을 읽어와서 실행하므로 오류가 발생하지 않는다.

( 버튼을 누를 때는 이미 한 번 렌더링을 해서 `this.scrollBox`를 설정한 시점이다. )

---

## 5.4 정리

컴포넌트 내부에서 DOM에 직접 접근해야 할 때는 `ref`를 사용한다.

`ref`를 사용하지 않고도 원하는 기능을 구현할 수 있는지 반드시 ! 고려한 후 활용해야한다.

다른 컴포넌트끼리 데이터를 교류할 때 `ref`를 사용할 수는 있다.

하지만, 앱 구조가 커질 경우 구조가 꼬여 버려서 유지 보수가 불가능한 상황이 발생할 수 있다.

따라서, 컴포넌트끼리 데이터를 교류할 때는 언제나 데이터를 `부모 <-> 자식` 흐름으로 교류해야한다.

(이것은 추후에 `Redux` 혹은 `Context API`를 사용하여 효율 적으로 교류하는 방법을 배울 것이다.)