1. Hooks 이전

- 컴포넌트 내부에 상태가 있다면?

\* class

- 컴포넌트 내부에 상태가 없다면?

\* 라이프사이클을 사용해야 한다면?

# class

\* 라이프사이클에 관계 없다면?

# function

2. Hooks 이후

- class

- function

3. Class 컴포넌트

import React from 'react';

//정의

class ClassComponent extends React.Component {

render() {

return (<div>Hello</div>);

}

}

//사용

<ClassComponent />

4. Function 컴포넌트

import React from 'react';

// 정의 1

function FunctionComponent() {

return <div>Hello</div>;

}

// 정의 2

const FunctionComponent = () => <div>Hello</div>;

//사용

<FunctionComponent/>

3. 실습

(1) Class 컴포넌트 만들기

//정의

class ClassComponent extends React.Component {

render() { //render 메서드 정의.

return <div>Hello</div>;

}

}

//사용

ReactDOM.render(

<ClassComponent />,

document.querySelector("#root")

);

(2) Function 컴포넌트 만들기

//정의 1

function FunctionComponent() {

return <div>Hello</div>;

}

// 정의 2

const FunctionComponent = () => <div>Hello</div>;

// 사용

ReactDOM.render(<FunctionComponent />, document.querySelector("#root"));

React.createElement(

type, // 태그 이름 문자열 | 리액트 컴포넌트 | React.Fragment

[props], // 리액트 컴포넌트에 넣어주는 데이터 객체

[...children] // 자식으로 넣어주는 요소들

)

(3)

1. 태그 이름 문자열 type

아래의 코드 = <h1>type 이 "태그 이름 문자열" 입니다.</h1>

ReactDOM.render(

React.createElement('h1', null, `type 이 "태그 이름 문자열" 입니다.`), // 순서 : 태그, 데이터, 자식

document.querySelector('#root')

);

2. 리액트 컴포넌트 type

const Component = () => { //(1) 상수 컴포넌트가 그려지면 실제로 React.createElement 안의 내용이 그려진다.

return React.createElement('p', null, `type 이 "React 컴포넌트" 입니다.`);

}

// 아래의 코드 = <Component></Component> => <Component /> => <p>type 이 "React 컴포넌트" 입니다.</p>

ReactDOM.render (

React.createElement(Component, null, null), //(2) (1)의 내용은 React.createElement()를 랜더 안에 넣어서 실제 돔에 그려졌을 때를 말한다.

document.querySelector("#root")

);

3. React.Fragment => 요소를 찍어봤을 때, 태그 없이 <div> 태그 안에 나온다.

React.Fragment가 필요한 이유는, 자식요소에 여러개를 넣을 수 있기 때문이다.

ReactDOM.render(

React.createElement(

React.Fragment,

null,

`type 이 "React Fragment" 입니다.`,

`type 이 "React Fragment" 입니다.`,

`type 이 "React Fragment" 입니다.`

),

document.querySelector("#root")

);

4. 복잡한 리액트 엘리먼트 모임

이 코드를 React.Element를 사용해서 만들기 위해서는..!

<h1>안녕하세요</h1>

<div>

<div>

<h1>주제</h1>

<ul>

<li>React</li>

</ul>

</div>

</div>

이런 식으로 쓸 수 밖에 없다. 하지만 이것은 매우 복잡한 방법이다.

ReactDOM.render(

React.createElement(

"div",

null,

React.createElement(

"div",

null,

React.createElement("h1", null, "주제"),

React.createElement(

"ul",

null,

React.createElement("li", null, "React"),

React.createElement("li", null, "Vue")

)

)

),

document.querySelector("#root")

)

그래서..

우리는 JSX를 사용한다.

5. JSX

JSX 문법으로 작성된 코드는 순수한 JavaScript로 컴파일하여 사용한다.

누가 컴파일을 해주나? -> babel이 해준다.

예)

//우리가 작성한 어떤 코드 => 순수하게 실행할 수 있는 자바스크립트

// babel

const title = "주제!!!";

ReactDOM.render(

<div a="a">

<div>

<h1>{title}</h1>

<ul>

<li>React</li>

<li>Vue</li>

</ul>

</div>

</div>,

document.querySelector("#root")

)

6. 왜 JSX를 쓰나요?

- React.createElement VS JSX => 가독성 완승

- babel과 같은 컴파일 과정에서 문법적 오류를 인지하기 쉬움.

7. JSX 문법

- 최상위 요소가 **하나**여야 합니다.

- 최상위 요소 리턴하는 경우, ()로 감싸야 합니다. => 요즘은 절대적인 규칙은 아니다.

- 자식들을 바로 랜더링하고 싶으면, **<>자식들</>** 를 사용합니다. => **Fragment**

- 자바스크립트 표현식을 사용하려면, **{표현식}**를 이용합니다.

- if 문은 사용할 수 없습니다. => 삼항 연산자 혹은 && 를 사용합니다.

- style을 이용해 인라인 스타일링이 가능합니다.

- class 대신 className 을 사용해 class를 적용할 수 있습니다.

- 자식요소가 있으면, 꼭 닫아야 하고, 자식요소가 없으면 열면서 닫아야 합니다.

=> <p>어쩌구</p>

=> <br/>

8. Props 와 State

- Props는 컴포넌트 외부에서 컴포넌트에게 주는 데이터입니다.

- State는 컴포넌트 내부에서 변경할 수 있는 데이터입니다.

- 둘 다 변경이 발생하면, 랜더가 다시 일어날 수 있습니다.

9. Render 함수

- Props와 State를 바탕으로 컴포넌트를 그립니다.

- 그리고 Props와 State가 변경되면, 컴포넌트를 다시 그립니다.

- 컴포넌트를 그리는 방법을 기술하는 함수가 랜더 함수입니다.

(1)

{message: '안녕하세요!!!'}

function Component(props) {

return (

<div>

<h1>{props.message} 이것은 함수로 만든 컴포넌트 입니다.</h1>

</div>

);

}

Component.defaultProps = {

message: "기본값!!!!",

};

(2)

class Component extends React.Component {

// state = { // state는 객체 형태로만 제공이 된다.

// count: 0,

// };

constructor(props) {

super(props);

this.state = {count: 0};

}

render() {

return (

<div>

<h1>{this.props.message} 이것은 클래스로 만든 컴포넌트 입니다.</h1>

<p>{this.state.count}</p>

</div>

);

}

(나머지)

// setState를 사용하면, 값이 바뀐 상태로 render 메소드에 자동으로 호출된다.

componentDidMount() {

setTimeout(() => {

// 객체를 통째로 새로 만들어서 넣는 방식

// this.state.count = this.state.count + 1;

// this.setState({

// count: this.state.count + 1,

// });

//앞의 값을 이용해서 할 수 있음.

this.setState((previousState) => {

const newState = {count: previousState.count + 1};

return newState;

})

}, 1000);

}

static defaultProps = {

message: "기본값2",

};

}

ReactDOM.render(

<Component message="기본값 아님"/>,

document.querySelector("#root")

);

10. Event Handling

- HTML DOM 에 클릭하면 이벤트가 발생하고, 발생하면 그에 맞는 변경이 일어나도록 해야합니다.

- JSX 에 이벤트를 설정할 수 있습니다.

- camelCase로만 사용할 수 있습니다. => onClick, onMouseEnter

- 이벤트에 연결된 자바스크립트 코드는 함수입니다. => 이벤트={함수} 와 같이 씁니다.

- 실제 DOM 요소들에만 사용 가능합니다. => 리액트 컴포넌트에 사용하면, 그냥 props로 전달합니다.

(1)

function Component() {

return (

<div>

<button

onClick={() => {

console.log('clicked');

}}

>

클릭

</button>

</div>

);

}

(2)

class Component extends React.Component {

state = {

count: 0,

};

// constructor(props) {

// super(props);

// this.click = this.click.bind(this);

// }

render() {

return (

<div>

<p>{this.state.count}</p>

<button onClick={this.click}>클릭</button>

</div>

);

}

click = () => {

console.log("clicked");

this.setState((state) => ({

...state,

count: state.count + 1,

}))

}

}

(나머지)

ReactDOM.render(<Component />, document.querySelector('#root'))

11. Component Lifecycle

- 리액트 컴포넌트는 탄생부터 죽음가지 여러지점에서 개발자가 작업이 가능하도록 메서드를 오버라이딩 할 수 있게 해준다.

- Declarative : 선언적으로 라이프사이클을 그려놓는 것.

\* Initialization

\* Mounting

\* Updation : props나 states가 변경됨.

\* Unmounting

- Component 생성 및 마운트 (< v16.3)

\* constructor

\* componentWillMount

\* render (최초 랜더)

\* componentDidMount

- Component props, state 변경 (<v16.3)

\* componentWillReceiveProps => props가 변경될 때 실행. (state가 변경될때는 제외.)

\* shouldComponentUpdate

\* componentWillUpdate

\* render

\* componentDidUpdate

- componentWillReceiveProps

\* props를 새로 지정했을 때 바로 호출됩니다.

\* 여기는 state의 변경에 반응하지 않습니다.

=> 여기서 props의 값에 따라 state를 변경해야 한다면,

==> setState를 이용해 state를 변경합니다.

==> **그러면 다음 이벤트로 각각 가는 것이 아니라 한번에 변경됩니다.**

- shouldComponentUpdate

\* props만 변경되어도 실행.

\* state만 변경되어도 실행.

\* props & state 둘 다 변경되어도 실행.

\* newProps와 newState를 인자로 해서 호출.

\* return type이 boolean 이다.

=> true 면 render

=> false 면 render가 호출되지 않는다.

=> 이 함수를 구현하지 않으면, 디폴트는 true 이다.

- componentWillUpdate

\* 컴포넌트가 재 랜더링 되기 직전에 불린다.

\* 여기서는 setState 같은 것을 쓰면 안됨.

- componentDidUpdate

\* 컴포넌트가 재 랜더링을 마치면 불린다.

- Component 언마운트 (< v16.3)

\* componentWillUnmount -> 언마운트가 되면 render를 할 수 없기 때문에, 언마운트 되기 직전에 무언가를 할 수 있는 컴포넌트이다. (메모리 정리, API 요청 시 컴포넌트를 받기 전에 언마운트가 된다면 더이상 받지 않겠다 등의 처리를 함.)

- 실습

class App extends React.Component {

state = {

age: 39,

};

interval = null;

constructor(props) {

super(props);

console.log("constructor", props);

}

//콘솔로 찍어봤을 때, 순서가 componentWillMount -> render -> componentDidMount 가 된다.

render() {

console.log("render");

return (

<div>

<h2>Hello {this.props.name} - {this.state.age}</h2>

</div>

);

}

componentWillMount() {

console.log("componentWillMount");

}

componentDidMount() { //타이머 처리, API 요청 등의 기능을 하게 됨.

console.log("componentDidMount");

this.interval = setInterval(() => {

// console.log("setInterval");

this.setState(state => ({...state, age: state.age+1}))

}, 1000);

}

componentWillReceiveProps(nextProps) {

console.log('componentWillReceiveProps', nextProps);

}

shouldComponentUpdate(nextProps, nextState) {

console.log('shouldComponentUpdate', nextProps, nextState);

return true; //return을 false로 한다면 다음으로 넘어가지 않기 때문에 render되지 않는다.

}

componentWillUpdate(nextProps, nextState) {

console.log('componentWillUpdate', nextProps, nextState);

}

componentDidUpdate(prevProps, prevState) {

console.log('componentDidUpdate', prevProps, prevState);

}

componentWillUnmount() {

clearInterval(this.interval);

}

}

ReactDOM.render(<App name="Mark" />, document.querySelector("#root"));

12. Component 라이프사이클 변경 (v16.3)

- constructor

- ~~componentWillMount~~ => getDerivedStateFromProps (앞에 static 키워드를 붙여야 함.)

- render

- componentDidMount

- ~~componentWillReceiveProps~~ => getDerivedStateFromProps (앞에 static 키워드를 붙여야 하며, return시에는 null로 리턴해야함. (리턴은 필수.) 특수한 경우에만 쓰임. 시간의 흐름에 따라 변경되는 props에 state가 의존하는 경우를 위해서 존재함.)

- shouldComponentUpdate

- render

- ~~componentWillUpdate~~ => getSnapshotBeforeUpdate (dom 적용 직전) (앞에 static 키워드를 붙여야 함.)

- (dom에 적용)

- componentDidUpdate (dom 적용 직후)

- componentWillUnmount

**- 실습1**

class App extends React.Component {

state = {

age: 39,

};

interval = null;

constructor(props) {

super(props);

console.log("constructor", props);

}

//콘솔로 찍어봤을 때, 순서가 componentWillMount -> render -> componentDidMount 가 된다.

render() {

console.log("render");

return (

<div>

<h2>Hello {this.props.name} - {this.state.age}</h2>

</div>

);

}

// static getDerivedStateFromProps(nextProps, prevState) {

// console.log("getDerivedStateFromProps", nextProps, prevState);

// return {

// age: 390,

// };

// }

componentDidMount() { //타이머 처리, API 요청 등의 기능을 하게 됨.

console.log("componentDidMount");

this.interval = setInterval(() => {

// console.log("setInterval");

this.setState(state => ({...state, age: state.age+1}))

}, 1000);

}

// componentWillReceiveProps(nextProps) {

// console.log('componentWillReceiveProps', nextProps);

// }

shouldComponentUpdate(nextProps, nextState) {

console.log('shouldComponentUpdate', nextProps, nextState);

return true; //return을 false로 한다면 다음으로 넘어가지 않기 때문에 render되지 않는다.

}

// componentWillUpdate(nextProps, nextState) {

// console.log('componentWillUpdate', nextProps, nextState);

// }

componentDidUpdate(prevProps, prevState) {

console.log('componentDidUpdate', prevProps, prevState);

}

componentWillUnmount() {

clearInterval(this.interval);

}

}

ReactDOM.render(<App name="Mark" />, document.querySelector("#root"));

**- 실습2**

let i = 0;

class App extends React.Component {

state = {list: []};

render() {

return (

<div id="list" style={{height:100, overflow:"scroll"}}>

{this.state.list.map((i) => {

return <div>{i}</div>

})}

</div>

);

}

componentDidMount() {

setInterval(() => {

this.setState(state => ({

list: [...state.list, i++],

}))

}, 1000);

}

getSnapshotBeforeUpdate(prevProps, prevState) {

if (prevState.list.length === this.state.list.length) return null; // return이 update 하기 전에 snapshot을 저장한다는 의미인데, return null은 아무것도 저장하지 않겠다는 의미이다.

const list = document.querySelector("#list");

return list.scrollHeight - list.scrollTop;

}

//숫자가 늘어날 때 마다 scorll도 같이 움직이게 하는 역할.

componentDidUpdate(prevProps, prevState, snapshot) {

console.log(snapshot);

if (snapshot === null) return;

const list = document.querySelector("#list");

list.scrollTop = list.scrollHeight - snapshot;

}

}

ReactDOM.render(<App name="Mark" />, document.querySelector("#root"));

13. Component 에러 캐치

- componentDidCatch : 자기 자신에게 문제가 있을 때 catch가 되지 않는다는 단점이 있다.

- Error Boundaries : 서비스(컴포넌트)의 최상위에 있어야 함. (하단에서 발생하는 에러를 캐치함.) 부모로 만들고, 하위 바운더리에 문제가 있을 때 에러를 캐치하는 방식으로 사용함.

**- 실습**

class App extends React.Component {

state = {

hasError: false

};

render() {

if (this.state.hasError) {

return <div>예상치 못한 에러가 발생했다.</div>

}

return <WebService />;

}

componentDidCatch(error, info) {

this.setState({hasError: true});

}

}

ReactDOM.render(<App name="Mark" />, document.querySelector("#root"));