Component (Intermediate)

이벤트 처리

- 이벤트 이름을 **카멜케이스(camelCase)**로 입력

```
// 기존의 DOM 이벤트 연결
<button onclick="activateLasers()">Activate Lasers</button>

// onclick이 아니라 onClick으로 입력 (DOM의 이벤트와 구분하기 위해서 카멜케이스 사용)
// 함수를 호출하는 JS 구문이 아닌, 함수 자체를 전달해야 함을 유의!
<button onClick={activateLasers}>Activate Lasers</button>
```

- 보통 관용적으로 이벤트 핸들러 이름 앞에 **handle이라는 접두어**를 붙임 (필수는 아님)

기본 동작 제거

- 리액트에서는 이벤트 핸들러 함수에서 false 값을 반환하여 기본 동작(링크 이동, 폼 전송 등)을 막도록 하는 방법을 사용할 수 없음

```
<a href="#" onclick="console.log('The link was clicked.'); return false">Click me</a>
```

- 대신, **이벤트 객체의 preventDefault 메소드를 호출**하여 기본 동작을 수행하지 않도록 조정

```
function ActionLink() {
    function handleClick(e) {
        // 이벤트 객체의 preventDefault 메소드 호출하여 기본 동작(링크로 이동)을 막기
        e.preventDefault();
        console.log('The link was clicked.');
    }

    return <a href="#" onClick={handleClick}>Click me</a>;
}
```

this 바인딩 문제 해결

- 자바스크립트의 this 바인딩 문제는 리액트 컴포넌트를 정의하는 과정에서도 발생 가능
  - https://ko.javascript.info/call-apply-decorators
  - https://ko.javascript.info/bind

```
js-study/this-study-1.js
```

```
"use strict"

const user = {
    func() { console.log('this :', this) }
}
const another = { name: 'another' }

// 함수 호출 시점에 메소드를 호출하는 주체에 따라서 this 값 바인딩
// 여기서는 user 객체가 this 값으로 바인딩 됨
user.func() // this : user 객체

const f = user.func
// 이 시점에서는 호출하는 주체가 없으므로 this는 undefined (엄격 모드가 아니라면 글로벌 객체)
f() // this : undefined

another.f = user.func
// 호출하는 주체가 another 객체로 바뀌었으므로 여기서는 another 객체가 this 값으로 바인딩 됨
another.f() // this : another 객체

// bind 메소드 바인딩을 통해 명시적으로 user 객체가 this 값으로 바인딩되도록 설정
const binded = user.func.bind(user)
// 비록 호출하는 주체는 없지만 앞서 명시적으로 this 바인딩을 수행했으므로 this는 user
binded() // this : user 객체
```

```
js-study/this-study-2.js
```

```
class MyClass {
    handleSomething() { console.log('this :', this) }
    someArrowFunc = () => { console.log('this :', this) }
}

const c = new MyClass()
console.log('c.handleSomething()')
// 호출하는 주체가 c 이므로 this는 c
c.handleSomething()

function func(handler) {
    console.log('handler()')
    handler()
}
// 이 경우 func 내부에서 함수를 호출하고 따로 주체가 없으므로 this는 undefined
func(c.handleSomething)

console.log('bind 작업 진행')
const binded = c.handleSomething.bind(c)
// func 내부에서 함수를 호출하고 따로 주체는 없지만, 명시적으로 this를 c로 바인딩 했으므로 this는 c
func(binded)

console.log('화살 함수 전달')
// 화살표 함수 전달했으므로 this는 c
```

```
// Node 12버전부터 실험적인 문법 기능 제공 (https://github.com/tc39/proposal-class-fields)
func(c.someArrowFunc)
```

- 핸들러 메소드 내부에서 (가령, setState 메소드 호출을 위해서) this 값을 참조해야 할 경우 **this 바인딩이 이루어지지 않아 undefined 값을 통해 메소드를 호출하는 문제**가 발생함
  - 이유) onClick과 같은 속성을 통해 전달된 핸들러 메소드가 호출될 때 명시적인 this(implicit this)를 확인할 수 없는 상태에서 호출이 이루어지고, 코드가 엄격 모드에서 실행되므로 this에는 undefined 값이 할당됨
- 이 문제를 해결하는 방법은 3가지
  - 생성자(constructor) 내부에서 **명시적으로 핸들러 메소드의 bind 메소드를 호출**하여 this 바인딩 작업을 진행
  - (권장) **화살표 함수를 이용**하여 컴포넌트 클래스를 정의하는 시점에 컴포넌트 객체로 this 바인딩이 진행되도록 유도
    - 단, 이 방법은 자바스크립트 언어에서 지원하는 정식 문법이 아님!
  - https://github.com/tc39/proposal-class-fields
  - 이벤트 바인딩 과정에서 **화살표 함수를 전달하고 해당 함수 내부에서 컴포넌트의 이벤트 핸들러 메소드 호출**
  - 단, 이 방법은 render 함수가 호출될때마다 새로운 화살표 함수 객체가 생성된다는 단점이 있음
- 컴포넌트에 미리 정의된 메소드(ex: 생명주기 메소드)들은 바인드 작업을 내부적으로 해주기 때문에 직접 정의한 핸들러 메소드만 바인딩 필요

해결 방법 코드

- **해결 방법 1** => 생성자 메소드 내부에서 bind 메소드 호출하여 this 바인딩
  - 단점 : 새로운 이벤트 핸들러 메소드를 추가할 때마다 매번 bind 메소드 호출을 해줘야 함

```
class Toggle extends React.Component {
    constructor(props) {
        super(props);
        this.state = {isToggleOn: true};

        // 콜백에서 this가 작동하게 하기 위해서 아래와 같이 bind 메소드를 이용하여 직접 바인딩
        this.handleClick = this.handleClick.bind(this);
    }

    handleClick() {
        // 기본적으로 내부 this가 컴포넌트 객체를 가리키지 않음
        this.setState(state => ({
            isToggleOn: !state.isToggleOn
        }));
    }

    render() {
        return (
            <button onClick={this.handleClick}>
                {this.state.isToggleOn ? 'ON' : 'OFF'}
            </button>
        );
```

```
    }
}
```

- **해결 방법 2** => 이벤트 바인딩 과정에서 화살표 함수 생성하여 전달
  - 단점 : onClick 내부의 JS 코드를 평가하는 과정에서 매번 새 함수 객체가 생성됨

```
class LoggingButton extends React.Component {
  handleClick() {
    console.log('this is:', this);
  }

  render() {
    // 이 문법은 this가 handleClick 내에서 바인딩되도록 합니다.
    return (
      <button onClick={() => this.handleClick()}>
        Click me
      </button>
    );
  }
}
```

- **(권장) 해결 방법 3 => 화살표 함수로 이벤트 핸들러 정의**
  - 단점 : 정식 문법이 아니라 실험적으로 지원하는 문법임
    - 원래 화살표 함수는 this 바인딩을 하지 않는 것이 원칙이지만, 여기서는 클래스 객체를 this로 바인딩

```
class LoggingButton extends React.Component {
    // 화살표 함수를 이용하여 this가 LoggingButton 컴포넌트를 가리키도록 설정
    handleClick = () => {
        console.log('this is:', this);
    }

    render() {
        return <button onClick={this.handleClick}>Click me</button>;
    }
}
```

이벤트 핸들러에 인자 전달

- 이벤트 핸들러에 **전달해야 할 인자가 있을 경우** 두 가지 방법 사용 가능
- 화살표 함수를 새로 정의하는 과정에서 해당 함수 내부에서 특정 인자값을 전달하도록 하기

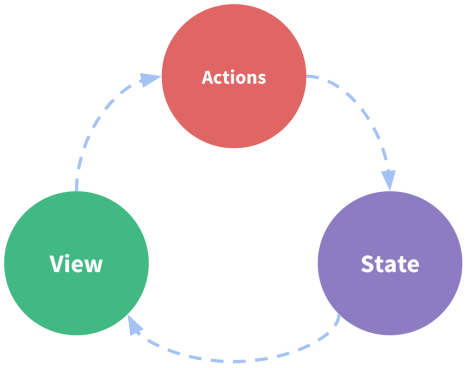
```
// (전달된 e는 이벤트 객체)
<button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>
```

- bind 메소드를 호출하여 컴포넌트 객체를 this로 바인딩하는 과정에서 추가 인자값을 전달하기

```
<button onClick={this.deleteRow.bind(this, id)}>Delete Row</button>
```

단방향 데이터 흐름

- 리액트는 **단방향 데이터 흐름(Unidirectional Data Flow, One-way binding)을 지향** (Vue의 경우 양방향 데이터 흐름을 지원)
- **한 쪽 방향(부모=>자식)**으로 변경된 상태값이 전달되며 컴포넌트 re-render 진행



- 뷰를 통해 상태 변경 진행 => 상태가 변경되었으므로 **부모 및 자식 컴포넌트가 반응(reaction)**하여 변경 사항 전달(action) 및 re-render => 다시 뷰가 구성되고 또 뷰를 통해 상태 변경 진행 => 반복...
    - 자식 컴포넌트에서 부모 컴포넌트의 상태를 변경할때에는 **부모 컴포넌트로부터 props 값을 통해 전달받은 핸들러 메소드를 이용**하여 수행
- 전반적인 단방향 데이터 흐름 정리 (*)
    1. 부모 컴포넌트는 **props를 통해서 ①자식 뷰를 구성하기 위한 정보와 필요한 경우 ②부모 컴포넌트 내부 상태를 변경할 수 있는 핸들러 메소드**를 전달함
    2. props를 전달받아 render만 수행하는 자식 컴포넌트는 그냥 뷰만 구성 (수동적인 역할 수행)
    3. props를 통해 핸들러 메소드를 전달받아 **부모 컴포넌트의 상태를 변경할 수 있는 자식 컴포넌트**의 경우 이벤트 바인딩 및 핸들러 메소드를 호출을 통해 부모 컴포넌트의 상태를 변경 가능
        - 부모 컴포넌트에서는 setState를 호출하여 내부 상태를 변경하고, 이로 인하여 부모 컴포넌트의 render 메소드가 호출되는 과정에서 자식 컴포넌트에 새로 props 값을 전달하므로 해당 부모 컴포넌트에 속한 모든 자식 컴포넌트의 re-render 작업이 수행

상위 컴포넌트로 상태 끌어올리기 (Lifting State Up)

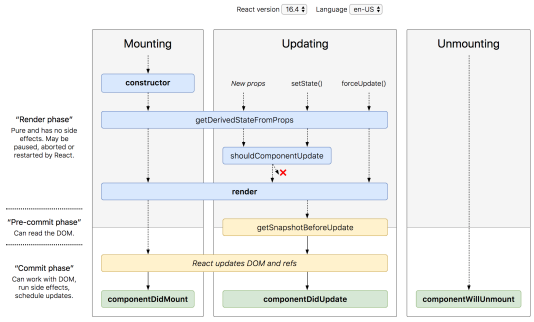
- 어떤 특정 상태 값에 관심있는 컴포넌트가 여럿 있는 경우 **해당 컴포넌트들의 가장 가까운 공통 조상 컴포넌트에 상태를 배치**하자는 전략
    - Often, several components need to reflect the same changing data. We recommend **lifting the shared state up to their closest common ancestor**.
    - ex) 할 일 관리 앱
        - TodoApp에서 목록 데이터(상태)는 TodoList가 아닌, TodoApp에 속해 있어야 함
        - 왜냐하면 TodoAdder도 목록 데이터를 변경해야 할 책임이 있고 상태 값에 관심이 있는 컴포넌트이므로, TodoList, TodoAdder의 공통 조상인 TodoApp에 상태가 위치하도록 상태를 올려(lifting state up)야 함
- https://ko.reactjs.org/docs/lifting-state-up.html
    - 관련있는 상태는 될 수 있으면 가까이에 보관하기 바랍니다. 애플리케이션의 특정 부분에만 상태가 필요하다면 **그 상태는 애플리케이션의 가장 높은 계층에 저장할 것이 아니라 최소 공통 부모 컴포넌트에서 관리**해야 합니다.

component-intermediate/lifting-state-up.js

```
import React from 'react'
import ReactDOM from 'react-dom'

const scaleNames = {
    c: 'Celsius',
    f: 'Fahrenheit'
};

function toCelsius(fahrenheit) {
    return (fahrenheit - 32) * 5 / 9;
}

function toFahrenheit(celsius) {
    return (celsius * 9 / 5) + 32;
}

function tryConvert(temperature, convert) {
    const input = parseFloat(temperature);
    if (Number.isNaN(input)) {
        return '';
    }
    const output = convert(input);
    const rounded = Math.round(output * 1000) / 1000;
    return rounded.toString();
}

function BoilingVerdict(props) {
    if (props.celsius >= 100) {
        return <p>The water would boil.</p>;
    }
    return <p>The water would not boil.</p>;
}

class TemperatureInput extends React.Component {
    constructor(props) {
        super(props);
        this.handleChange = this.handleChange.bind(this);
    }

    handleChange(e) {
        this.props.onTemperatureChange(e.target.value);
    }

    render() {
        const temperature = this.props.temperature;
        const scale = this.props.scale;
        return (
            <fieldset>
                <legend>Enter temperature in {scaleNames[scale]}:</legend>
                <input value={temperature}
                       onChange={this.handleChange} />
            </fieldset>
```

```
        );
    }
}

class Calculator extends React.Component {
    constructor(props) {
        super(props);
        this.handleCelsiusChange = this.handleCelsiusChange.bind(this);
        this.handleFahrenheitChange =
this.handleFahrenheitChange.bind(this);
        this.state = {temperature: '', scale: 'c'};
    }

    handleCelsiusChange(temperature) {
        this.setState({scale: 'c', temperature});
    }

    handleFahrenheitChange(temperature) {
        this.setState({scale: 'f', temperature});
    }

    render() {
        const scale = this.state.scale;
        const temperature = this.state.temperature;
        const celsius = scale === 'f' ? tryConvert(temperature, toCelsius) :
temperature;
        const fahrenheit = scale === 'c' ? tryConvert(temperature,
toFahrenheit) : temperature;

        return (
            <div>
                <TemperatureInput
                    scale="c"
                    temperature={celsius}
                    onTemperatureChange={this.handleCelsiusChange} />
                <TemperatureInput
                    scale="f"
                    temperature={fahrenheit}
                    onTemperatureChange={this.handleFahrenheitChange} />
                <BoilingVerdict
                    celsius={parseFloat(celsius)} />
            </div>
        );
    }
}

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

생명주기 메소드



constructor (*)

- 컴포넌트를 **생성하는 시점에 단 한 번만 호출** (실제로 mount되기 전 시점(즉, DOM에 삽입되기 전)에 호출됨)
- **this.state 객체의 초기화 작업이 이루어지는 메소드**
  - 객체를 대입하며 상태 초기값, 기본값을 지정
  - **생성자 메소드 내부에서는 setState를 호출하지 말아야 함**, 그냥 객체 초기화를 하는 시점에 값을 설정해주면 됨 (반대로, 생성자 메소드가 아닌 메소드에서는 setState를 호출해서 상태를 변경해야 함)
- 화살표 함수를 쓰지 않는 경우 이벤트 핸들러의 this 바인딩을 생성자에서 수행 가능

```
constructor(props) {
    super(props);
    // 생성자 메소드 내부에서 this.setState() 메소드 호출 금지! (그냥 다음과 같이 state 객체 초기화)
    this.state = { counter: 0 };
    // 이벤트 핸들러의 this 바인딩이 필요한 경우 작업 진행
    this.handleClick = this.handleClick.bind(this);
}
```

- 다음과 같이 생성자가 아닌 **클래스 변수를 선언하는 방식으로 state 초기화 가능**

```
class MyComponent extends Component {
    // 클래스 변수 형태로 state 객체 정의
    state = {
        counter: 0,
    };
    // ...
}
```

- 생성자 내부에서 전달받은 props 값을 super 함수를 통해서 상속을 해준 클래스(React.Component)로 전달해야 함! (비록 **전달받은 props 값이 없다고 하더라도 해당 구문 호출하는 것이 권장**됨)

```
class MyComponent extends Component {
    constructor(props) {
        // super 함수 호출하며 props 값 전달
        super(props);
        // state 객체값 초기화
        this.state = {
            counter: 0,
        };
    }
}
```

- 전달받은 props 값을 이용하여 컴포넌트의 상태를 초기화하지 않도록 주의!

```
constructor(props) {
    super(props);

    // 이럴 필요 없음 (그냥 내부적으로 color값이 필요하면, this.props.color를 사용하면 됨)
    this.state = { color: props.color };
}
```

render (**)

- 뷰로 사용할 **JSX를 구성하고 반환**하는 역할 수행
- 보통 props, state 값에 따라 뷰의 내용 혹은 구성을 변경함
    - 따라서, 부모 컴포넌트로 부터 전달받은 **props 값이 변경될 경우 호출되거나 setState 통해서 상태가 변경**된 경우 render 메소드가 다시 호출됨

componentDidMount (*)

- 컴포넌트가 DOM에 **mount된 이후 (생애주기 내에서) 단 한 번만 호출** (즉, 최초로 render 메소드가 한 번 호출되어 그려진 이후, componentDidMount 메소드가 호출됨)
- 주로 사용되는 용도
    - **네트워크 요청(ajax 요청) 보내기 (*)**
    - 단, 네트워크 요청이 **컴포넌트 마운트 시점에 모두 완료된다는 보장이 없으므로** 상태(loading)를 추가한다던가 해서 (render 메소드 내부에서) **요청이 이루어지고 있는 시점에 보여줄 뷰와 요청이 성공 혹은 실패한 시점에 보여줘야 할 뷰를 따로 구성**해야 함
        - 보통 로딩 중인지 알려주는 상태값(loading)과 성공, 실패 여부를 알려줄 상태값(sucess)을 추가
            - 실패할 경우 이유를 알려줘야 할 수도 있으므로 상태값(msg, reason, error 등) 추가 가능
    - 타이머(setTimeout, setInterval) 시작하기
- 구독 요청 보내기 (ex: 파이어베이스 메시징)
    - side-effect가 발생하는 작업 수행 => DOM 직접 조작 (ex: 캔버스 태그 조작, DOM 요소의 native 메소드(스크롤 조작, 포커스 주기) 호출, 직접 addEventListener 메소드 호출하여 이벤트 리스너 설정하는 경우)
    - 여기에는 외부 라이브러리와의 연동 작업도 포함됨 (D3, Masonry 등)

componentDidUpdate

```
componentDidUpdate(prevProps, prevState)
```

- props, state 값이 변경된 이후 시점에 호출
  - componentDidUpdate() is invoked immediately after updating occurs. This method is not called for the initial render.
- 첫 render 메소드가 호출된 시점 이후(두 번째 render 호출 이후부터)에 호출
  - 첫 render 메소드 호출된 시점 이후에는 componentDidMount 메소드가 호출됨 (그림 참조)
  - 첫 render 메소드 => componentDidMount => props, state 업데이트 => 두 번째 render 메소드 => componentDidUpdate 호출 => 이후 props, state 업데이트에 따르는 render 메소드 호출 이후 계속해서 호출
    - 생애 주기 동안 한 번만 호출되는 componentDidMount 메소드와는 달리 componentDidMount는 상태 변화가 있을 때마다 계속 호출됨을 유의!
- 가령, state 값이 바뀔 때(ex : input 창의 입력값, count 함수)마다 네트워크 요청을 보낼 일이 있는 경우 사용 가능

componentWillUnmount (*)

- 컴포넌트가 **DOM에서 삭제되기 직전 (생애주기 내에서) 단 한 번만 호출**
  - 타이머 해제, 네트워크 요청 취소, 구독 취소 등 (componentDidMount에서 한 작업 반대로 하기)
    - C++의 파괴자, 안드로이드의 onDestory와 비슷한 역할 수행 (더 이상 사용되지 않는 리소스 청소 작업)
- DOM에서 노드 자체가 사라질 때 호출되며 display: none, visibility: hidden과 같이 **CSS 속성을 통해 화면에서 안 보이게 한다고 해서 호출되지는 않음**

```
component-intermediate/lifecycle-methods.js
```

```
import React, { Component } from 'react'
import ReactDOM from "react-dom"

class Container extends Component {
    constructor(props) {
        super(props)

        this.state = { prop1: 'prop1', prop2: 1, mount: true }
    }

    updateProp = () => {
        this.setState((state) => ({
            prop1: 'prop' + (state.prop2 + 1),
            prop2: state.prop2 + 1
        }))
    }

```

```
    render() {
        return (<div>
            <hr />
            {this.state.mount && <LifecycleMethodsDemo prop1={this.state.prop1} prop2={this.state.prop2} />}
            <hr />
            <button onClick={this.updateProp}>update prop</button>
            <button onClick={() => this.setState({ mount : true })}>mount</button>
            <button onClick={() => this.setState({ mount : false })}>unmount</button>
        </div>)
    }
}

class LifecycleMethodsDemo extends Component {
    // 컴포넌트가 생성되는 시점에 단 한 번 호출
    constructor(props) {
        super(props)

        console.log('constructor')

        this.state = { value: 0 }
    }

    // 컴포넌트의 첫 번째 render 호출으로 인한 mount 작업 이후 단 한 번 호출
    componentDidMount() {
        console.log('componentDidMount')
    }

    // props, state 변경시마다 render 함수 호출 이후 호출
    componentDidUpdate(prevProps, prevState) {
        console.log('componentDidUpdate', prevProps, prevState)
    }

    // unmount 시점에 호출
    componentWillUnmount() {
        console.log('componentWillUnmount')
    }

    // props, state 값 변경에 의해서 호출
    render() {
        return (
            <div>
                <p>state : {this.state.value}</p>
                <p>props : prop1 : {this.props.prop1} prop2 : {this.props.prop2}</p>
                <button onClick={() => this.setState((state) => {
                    return { value: state.value + 1 }
                })}>update</button>
            </div>
        );
    }
}

ReactDOM.render(<Container />, document.getElementById("root"))
```

Project) 타이머 컴포넌트

- componentDidMount, componentWillUnmount를 이용한 타이머 등록 및 해제

```
component-intermediate/timer-demo.js
```

```
import React, {Component} from 'react'
import ReactDOM from 'react-dom'

// Q) stop, resume 버튼 추가하기
class Timer extends Component {
    constructor(props) {
        super(props)

        this.state = {
            time: this.props.time,
            timeout: false,
            intervalId: null
        }
    }

    componentDidMount() {
        // 타이머 설정
        this.state.intervalId = setInterval(() => {
            this.setState((state) => {
                if( state.time === 1 ) {
                    clearTimeout(this.state.intervalId)
                    return { timeout: true, time: state.time - 1}
                } else {
                    return { time: state.time - 1 }
                }
            })
        }, 1000)
    }

    componentWillUnmount() {
        // 타이머 해제
        clearTimeout(this.state.intervalId)
    }

    render() {
        return (
            <div>
                {this.state.timeout ? <h2>timeout</h2> : <h2>
{this.state.time}</h2>}
            </div>
        );
    }
}

ReactDOM.render(
    <div>
        <Timer time={10} />
        <Timer time={30} />
```

```
        <Timer time={60} />
    </div>,
document.getElementById("root"))
```

shouldComponentUpdate

- props, state 값이 변경될 경우 render를 호출하기 전에 미리 호출됨
- 파라미터로 변경될 props, state 값이 전달됨

```
shouldComponentUpdate(nextProps, nextState)
```

- 해당 메소드에서는 불리언 값(true, false)을 반환해야 함
  - 기본 동작은 true를 반환
  - 만약 **true를 반환하면 render 함수를 호출**함 (**false를 반환하면 render를 호출하지 않음**)
- 보통은 props, state 값이 변하면 해당 값의 변화에 따라 뷰를 변화시켜야 하므로 render를 호출해야 함. 그래서 보통 shouldComponentUpdate 메소드를 재정의할 일은 잘 없음
  - 단, props, state 값이 변해도 render를 호출할 필요가 없는 경우(즉, 뷰를 재구성할 필요가 없는 경우) 내부적으로 props, state 값을 조회하여 특정 값이 변항 경우는 render 함수 호출을 하지 않도록 할 수 있음
    - this.props와 nextProps값을, this.state와 nextState 값을 비교
  - 결국 shouldComponentUpdate 메소드를 재정의하는 이유는 **퍼포먼스 최적화를 하기 위해서임**

```
component-intermediate/lifecycle-method-should-component-update.js
```

shouldComponentUpdate를 재정의한 컴포넌트

```
import React, { Component } from 'react'
import ReactDOM from 'react-dom'

class Container extends Component {
    constructor(props) {
        super(props)

        this.state = { count: 1 }
    }

    render() {
        return (<div>
            <button onClick={ () => this.setState(s => ({ count: s.count + 1 })) }>update count</button>
            <ShouldComponentUpdateDemo count={this.state.count} />
        </div>)
    }
}

class ShouldComponentUpdateDemo extends Component {
    constructor(props) {
        super(props)

```

```
        this.state = { text: "a" }
    }

    // props로 전달된 count 값이 짝수이거나 내부 상태값 중 text의 길이가 3의 배수가 될 경우 렌더링할 필요가 없다고 가정
    shouldComponentUpdate(nextProps, nextState, nextContext) {
        console.log(nextProps, nextState, nextContext)

        if((nextProps.count % 2) === 0) {
            console.log("count 값이 짝수이므로 렌더링하지 않음")
            return false
        } else if((nextState.text.length % 3) === 0) {
            console.log("text 길이가 3의 배수이므로 렌더링하지 않음")
            return false
        }

        return true
    }

    render() {
        console.log("render")

        return (<div>
            <p>count : {this.props.count}</p>
            <p>text : {this.state.text}</p>
            <button onClick={ () => this.setState(s => ({ text : s.text + "a" })) }>update text</button>
        </div>)
    }
}

ReactDOM.render(<Container />, document.getElementById("root"))
```

forceUpdate 메소드

- props, state 변화와 무관하게 **강제로 render 메소드를 호출**하길 원할 경우 forceUpdate 메소드 사용
- 가급적 사용 자제 (props, state 값의 변경에 따라 render 메소드가 잘 동작할 수 있도록 설계하는 것이 중요)

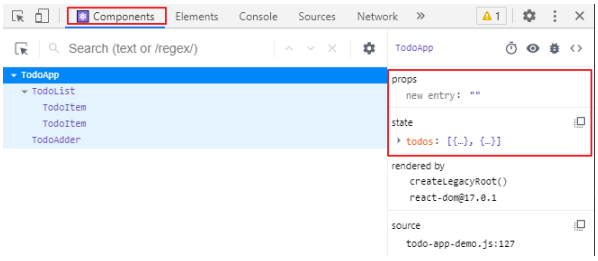
component-intermediate/force-update-demo.js

```
import React, { Component } from 'react'
import ReactDOM from 'react-dom'

class ForceUpdateDemo extends Component {
    constructor(props) {
        super(props)
        this.state = { intervalId: null }
    }

    componentDidMount() {
        this.state.intervalId = setInterval(() => {
```

```
            // forceUpdate 메소드 호출하여 강제로 render 호출
            console.log("forceUpdate")
            this.forceUpdate();
        }, 1000)
    }

    componentWillUnmount() {
        clearTimeout(this.state.intervalId)
    }

    render() {
        console.log("render")
        return <div>{ (new Date()).toISOString() }</div>
    }
}

ReactDOM.render(<ForceUpdateDemo />, document.getElementById('root'))
```

React Developer Tools 사용

- 크롬 확장 프로그램으로 제공되는 리액트 컴포넌트 모니터링 툴
  - React Developer Tools
- 프로그램 설치 후 개발자 도구의 Components 탭 접근
- 이후 특정 컴포넌트 선택하여 **관찰 시점의 props, state 값을 확인 가능**



리퍼런스

- https://reactjs.org/docs/state-and-lifecycle.html
- https://reactjs.org/docs/react-component.html
- https://developmentarc.gitbooks.io/react-indepth/content/life_cycle/introduction.html
- https://blog.bitsrc.io/react-16-lifecycle-methods-how-and-when-to-use-them-f4ad31fb2282