2nd React

ref

- DOM 을 꼭 직접적으로 건들여야 할 때, DOM 에 이름 다는 방법
- 전역적으로 작동하지 않고 컴포넌트 내부에서만 작동
- 특정 input에 포커스 주기/스크롤 박스 조작하기/Canvas 요소에 그림 그리기 등 state 만으로 해결할 수 없는 기능들을 위해 사용

▼ DOM 요소에 ref 달기

- 콜백 함수 `<input ref={(ref) ⇒ {this.input=ref}} />`또는 createRef 를 사용하여 ref 설정 가능
- 접근은 `this.input.current` 로 가능

▼ 컴포넌트에 ref 달기

- 컴포넌트 내부에 있는 DOM을 컴포넌트 외부에서 사용할 때 씀
- `<MyComponent ref={(ref)⇒ {this.myComponent=ref}} />`

ex) 스크롤 박스를 하나 만들고 스크롤바를 아래로 내리는 작업을 부모 컴포넌트에서 실행하기

- 함수형 컴포넌트 에서는 useRef 라는 hook 함수를 사용한다
- **서로 다른 컴포넌트 끼리 데이터를 교류할 때 쓰는 용도가 아니다.** (이는 리덕스 또는 Context API 를 활용, 부모 ↔ 자식 흐름으로 교류해야함)

컴포넌트의 반복

▼ map()

```
const numbers = [1, 2, 3, 4, 5];
const result = numbers.map(num ⇒ num * num);

const names = ['눈사람', '얼음', '눈', '바람'];
const nameList = names.map(name => <li>{name}</li>);
return <ul>{nameList}</ul>
```

```
// Warning: each child in a list should have a unique key prop.

const names = ['눈사람', '얼음', '눈', '바람'];
const nameList = names.map((name, index) => <li key={index}>{name}</li>);
return <ul>{nameList}</ul>
```

인덱스 값을 key로 사용하면 랜더링이 비효율적 → 그럼 어떻게 고유의 값을 만들까?

→ 배열 객체 안에 id 항목 추가

데이터 추가는 onClick 속성에 `배열.concat({  });` 사용

컴포넌트의 라이프사이클 메서드

▼ **마운트** (페이지에 컴포넌트가 나타남)

- constructor: 컴포넌트를 새로 만들 때마다 호출되는 클래스 생성자 메서드
- getDerivedStateFromProps: props 에 있는 값을 state에 넣을 때 사용하는 메서드
- render : UI를 렌더링
- componentDidMount: 컴포넌트가 웹 브라우저상에 나타난 후 호출하는 메서드

▼ **업데이트** (컴포넌트 정보를 업데이트)

1. props 가 바뀔때
2. state 가 바뀔때
3. 부모 컴포넌트가 리렌더링 될때
4. this.forceUpdate로 강제로 렌더링을 트리거할 때

- getDerivedStateFromProps: 마운트 과정에서 호출. 업데이트 시작 전 props에 변화에 따라 state 값에도 변화를 주고싶을 때 사용
- shouldComponentUpdate: 컴포넌트가 리렌더링을 해야 할지 말아야 할지를 결정하는 메서드. true 혹은 false 반환
- render: 컴포넌트를 리렌더링

- getSnapshotBeforeUpdate: 컴포넌트 변화를 DOM에 반영하기 바로 직전에 호출하는 메서드.
- componentDidUpdate: 컴포넌트의 업데이트 작업이 끝난 후 호출하는 메서드

▼ **언마운트** (페이지에서 컴포넌트가 사라짐)

componentWillUnmount: 컴포넌트가 웹 브라우저상에서 사라지기 전에 호출

라이프사이클 메서드 활용

▼ 예제

```
import React, { Component } from 'react';

class LifeCycleSample extends Component {
  state = {
    number: 0,
    color: null
  }

  myRef = null;

  constructor(props) {
    super(props);
  }

  // 부모에게서 받은 color값을 state에 동기화
  static getDerivedStateFromProps(nextProps, prevState) {
    if(nextProps.color != = prevState.color)
    {
      return { color: nextProps.color };
    }
  }

  shouldComponentDidUpdate(nextProps, nextState) {
    return nextState.number % 10 != =4;
  }

  // DOM 에 변화가 일어나기 직전의 색상 속성을 snapshot 값으로 반환하여
  // 이것을 componentDidUpdate에서 조회 가능하게 함
  getSnapshotBeforeUpdate(prevProps, prevState) {
    if (prevProps.color != = this.props.color) {
      return this.myRef.style.color;
```

```
    }
    return null;
  }
}

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

componentDidUpdate(prevProps, prevState, snapshot) {
  if(snapshot) {
    console.log('업데이트 되기 직전 색상: ', snapshot);
  }
}

render() {
  const style = {
    color: this.props.color
  };

  return (
    <div>
      <h1 style={style} ref={ref => this.myRef=ref}>
        {this.state.number}
      </h1>
      <p>color: {this.state.color}</p>
      <button onClick={this.handleClick}>
        더하기
      </button>
    </div>
  )
}
}
```