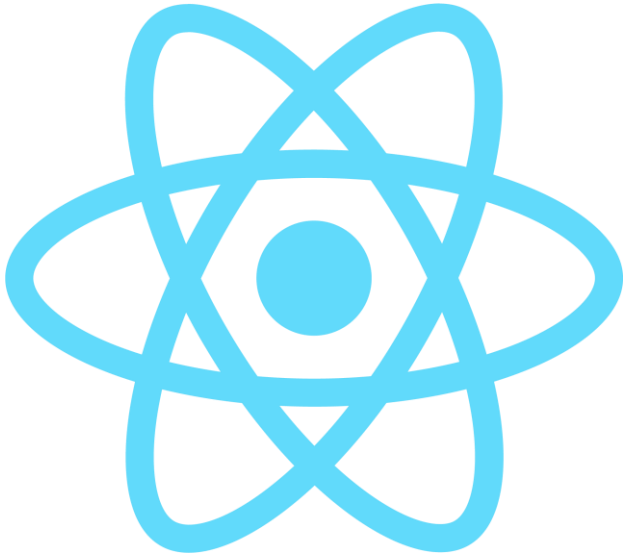
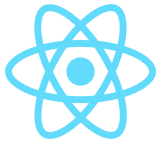


React
Main Concepts
07 - 12



조건부 렌더링

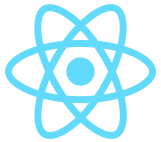
애플리케이션 상태에 따라서 일부 component를 렌더링

변수에 하나의 element를 저장할 수 있음

특정 조건을 만족할 때 element 변수를 선언해서 렌더링

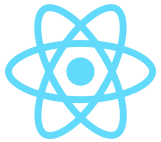
ex) 로그인 유무에 따라 다른 화면을 렌더링

&& 연산자, 삼항 조건 연산자를 활용해서 간결한 코드 가능



조건부 렌더링

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  let button;

  if (isLoggedIn) {
    button = <LogoutButton onClick={this.handleLogoutClick} />;
  } else {
    button = <LoginButton onClick={this.handleLoginClick} />;
  }

  return (
    <div>
      <Greeting isLoggedIn={isLoggedIn} />
      {button}
    </div>
  );
}
```

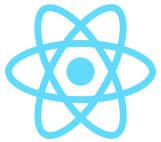


조건부 렌더링

특정 component 렌더링을 하지 않으려면 null을 반환

null 값인 component라도, 생명주기 메서드에 영향을 받음

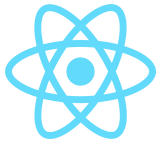
ex) componentDidUpdate 메서드가 여전히 작동



조건부 렌더링

```
function WarningBanner(props) {
    if (!props.warn) {
      return null;
    }
    return (
      <div className="warning">
        Warning!
      </div>
    );
  }
```

```
    render() {
      return (
        <div>
          <WarningBanner warn={this.state.showWarning} />
          <button onClick={this.handleToggleClick}>
            {this.state.showWarning ? 'Hide' : 'Show'}
          </button>
        </div>
      );
    }
```

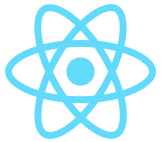


리스트와 key

map() 함수를 이용하여 인자를 적절하게 가공가능

map() 함수 결과를 저장한 element를 렌더링하면 항목이 알아서 구분됨

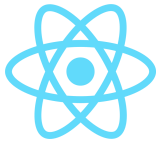
```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((numbers) =>
  <li>{numbers}</li>
);

ReactDOM.render(
  <ul>{listItems}</ul>,
  document.getElementById('root')
);
```



리스트와 key

리스트를 component에서 렌더링할 경우, 각 항목에 key 속성이 필요

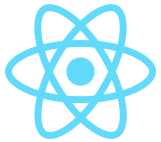
- 리스트의 각 항목에 접근할 때 key 속성값을 이용
- key 속성값으로 적절한 게 없다면 해당 항목의 index를 key로 사용하기도 함 (default)
- 다만 항목의 순서가 바뀔 수 있는 경우라면 index를 사용하지 않는 것이 좋음



리스트와 key

```
const listItems = numbers.map((number) =>
    <li key={number.toString()}>
        {number}
    </li>
);
```

```
const todoItems = todos.map((todo, index) =>
  // Only do this if items have no stable IDs
  <li key={index}>
    {todo.text}
  </li>
);
```

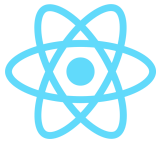


리스트와 key

key 속성은 배열이 위치한 context에서 유효함

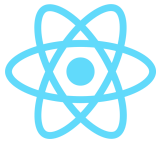
map() 함수 내부에 있는 element에 key 속성값을 넣어주는 걸 권장

key는 한 배열 안에서만 유효하기 때문에 다른 배열에서 map() 함수를 사용할 경우
key 값이 동일해도 무관함



리스트와 key

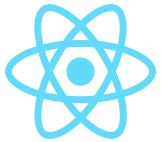
```
function ListItem(props) {
    // There is no need to specify the key here:
    return <li>{props.value}</li>;
}

function NumberList(props) {
    const numbers = props.numbers;
    const listItems = numbers.map((number) =>
        // Key should be specified inside the array.
        <ListItem key={number.toString()} value={number} />
    );
    return (
        <ul>
            {listItems}
        </ul>
    );
}
```



리스트와 key

element의 key 속성은 다른 component에 전달되지 않기 때문에 key 값을 전달하려면
다른 이름의 prop으로 전달해야 함

```
const content = posts.map((post) =>
  <Post
    key={post.id}
    id={post.id}
    title={post.title} />
);
```

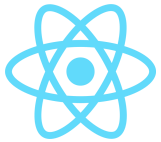


리스트와 key

map() 함수를 element 변수로 선언 후 사용하는 것 외에도, { }안에 포함하여 인라인 처리 가능

여러 map() 함수가 중첩된다면 가독성을 고려하여 component로 사용하는 걸 권장

```
const numbers = props.numbers;
return (
  <ul>
    {numbers.map((number) =>
      <ListItem key={number.toString()} value={number} />
    )}
  </ul>
);
```



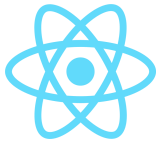
form (폼)

HTML의 기본 form 동작보다는, JS 함수로 form의 동작을 처리하고
user가 form에 입력한 데이터에 접근하는 방식을 이용

React에서는 component의 변경 가능한 값들이 state에 기록되므로
setState()로 값을 업데이트하도록 수정하여 이용

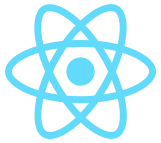
form에서 많이 사용하는 〈textarea〉, 〈input〉, 〈select〉 모두 유사하게 동작

여러 element를 사용할 때는 name 속성으로 구별



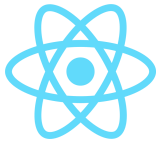
form (폼)

```
handleChange(event) {
  this.setState({value: event.target.value});
}

handleSubmit(event) {
  event.preventDefault();
}

render() {
  return (
    <form onSubmit={this.handleSubmit}>
      <label>
        Name:
        <input type="text" value={this.state.value} onChange={this.handleChange} />
      </label>
      <input type="submit" value="Submit" />
    </form>
  );
}
```



form (폼)

```
render() {
  return (
    <form>
      <label>
        Is going:
        <input name="isGoing" type="checkbox"
          onChange={this.handleInputChange} />
      </label>
      <label>
        Number of guests:
        <input name="numberOfGuests" type="number"
          onChange={this.handleInputChange} />
      </label>
    </form>
  );
}
```



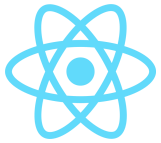
State 끌어올리기 (동기화)

여러 component에서 각각의 state 업데이트가 동기화되어 이뤄져야 하는 경우

한 component의 state는 해당 component에서만 유효하므로 동일한 값으로 유지되지 않음

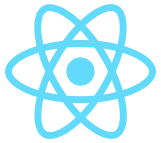
각 component의 동일한 부모 component로 state를 끌어올림으로써 동기화

이때 하위 component에서는 local state를 사용하기 보다
prop를 통해 값을 가져오고, 업데이트할 때는 부모 component의
handleChange 메서드를 호출하여 동기화



State 끌어올리기 (동기화)

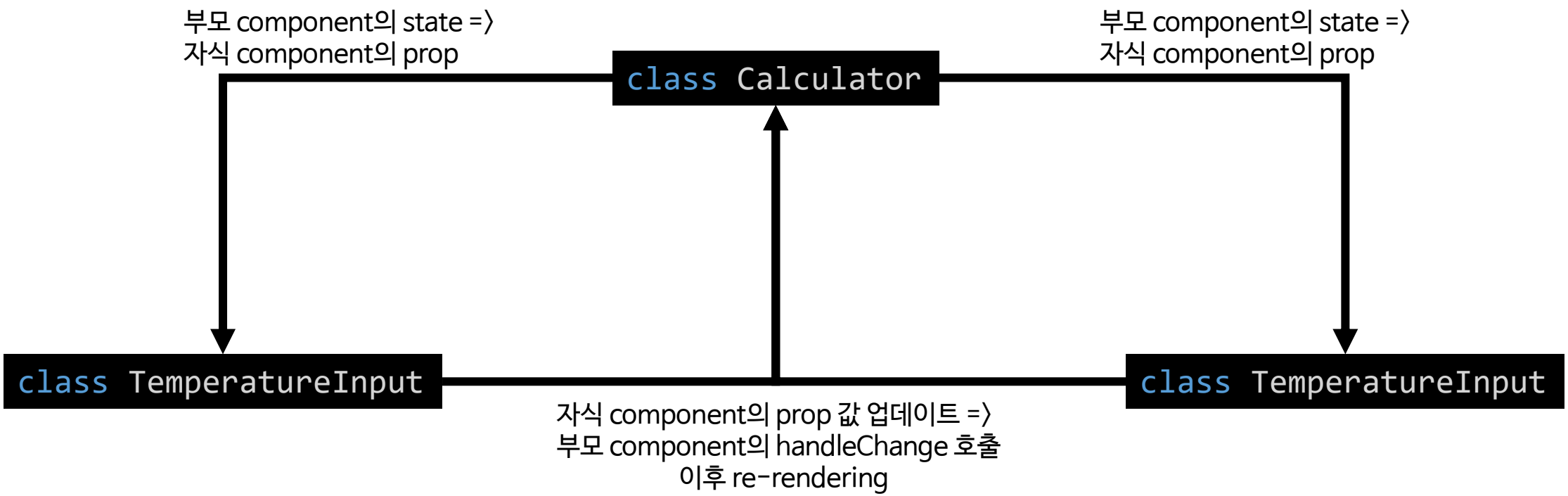
```
class Calculator extends React.Component {
  constructor(props) { ... }

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

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

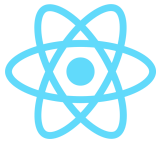


State 끌어올리기 (동기화)

```
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>
    );
  }
}
```

State 끌어올리기 (동기화)



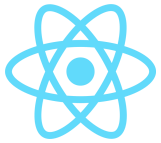


합성 (Composition) VS 상속 (Inheritance)

React는 상속 대신 합성을 사용하여 component 간에 코드를 재사용하는 걸 권장

한 component에서 다른 component를 포함하면서, 어떤 자식 component인 지 모르는 경우 prop의 children 속성을 사용하여 자식 element를 그대로 렌더링 가능

```
function FancyBorder(props) {
    return (
      <div>{props.children}</div>
    );
}

function WelcomeDialog() {
    return (
      <FancyBorder color="blue">
        <h1>Welcome</h1>
      </FancyBorder>
    );
}
```



합성 (Composition) VS 상속 (Inheritance)

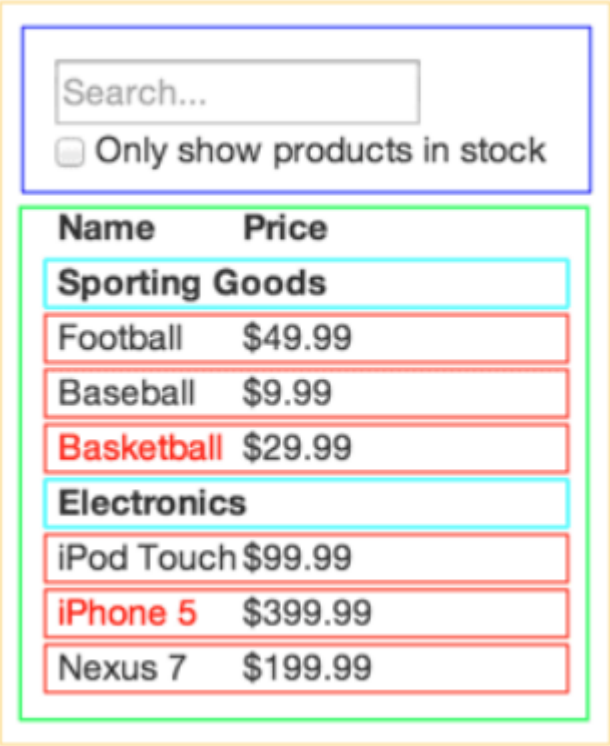
한 component에서 다른 component를 포함하면서, 상하 관계인 경우
구체적인 component가 일반적인 component를 렌더링 가능

```
function Dialog(props) {
    return (
      <FancyBorder color="blue">
        <h1>{props.title}</h1>
        <p>{props.message}</p>
      </FancyBorder>
    );
}

function WelcomeDialog() {
    return (
      <Dialog title="Welcome"
              message="Thank you for visiting our spacecraft!" />
    );
}
```

Build by React

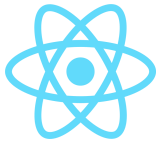
구현할 UI를 하나의 component로 구분

하나의 component는 한 가지 일을 하도록 분리 (단일 책임 원칙)



- 노랑: FilterableProductTable
- 파랑: SearchBar
- 연두: ProductTable
- 하늘: ProductCategoryRow
- 빨강: ProductRow

- FilterableProductTable
  - SearchBar
  - ProductTable
    - ProductCategoryRow
    - ProductRow



Build by React

간단한 UI인 경우 계층구조 상의 상층부 component부터 만드는 게 수월함

프로젝트 규모가 커지면 상향식으로 만들며 테스트하는 게 수월함

props와 state 용도 구분

props는 읽기 전용임을 고려하여 component 중, 데이터 변경이 필요 없는 곳에 이용

최상단 component의 데이터 모델이 변경되면 re-render되어 UI가 업데이트 됨 (단방향 흐름)

Build by React

상호작용 UI가 필요한 곳에는 state를 통해 업데이트

state 항목은 최소 집합으로 이루어져야 함 (중복 배제 원칙)

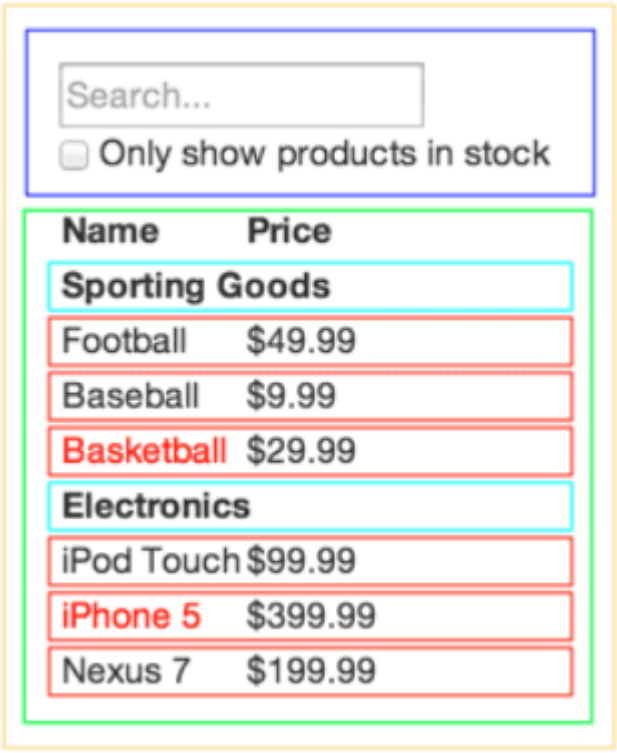
ex) 예시 애플리케이션에서는

- 제품의 원본 목록
- 유저가 입력한 검색어
- 체크박스 체크 유무
- 필터링 된 제품들의 목록

4가지 데이터를 갖고 있지만, 상호작용을 위한 데이터는

- 유저가 입력한 검색어
- 체크박스 체크 유무

2가지임



Build by React

어떤 component가 state를 필요로 하는 지 결정

state를 기반으로 렌더링하는 component를 찾고, 해당 component들의
공통 부모 component를 반복해서 찾아서 상위에 있는 component가 state를 갖도록 구현

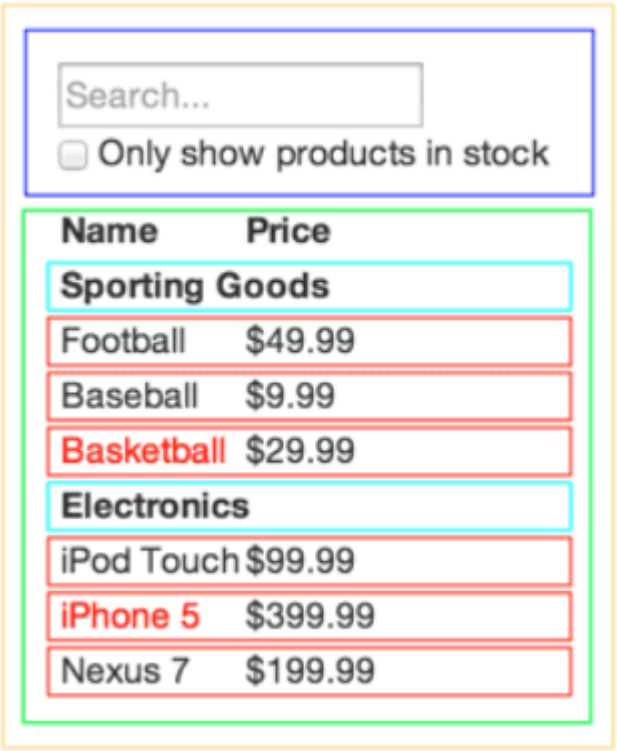
ex) 예시 애플리케이션에서는

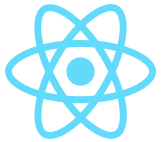
- ProductTable
- SearchBar

2가지 component가 state를 기반으로 렌더링하고,
두 component의 상위 component는

- FilterableProductTable

이므로 해당 component가 state를 갖도록 구현





Build by React

state 값 업데이트를 위한 역방향 흐름 추가

component는 자신의 local state만 업데이트할 수 있기 때문에
하위 component에 prop을 전달할 때, setState()를 호출하는
콜백을 넘겨서 state 값을 업데이트 하도록 구현

state 끌어올리기 활용