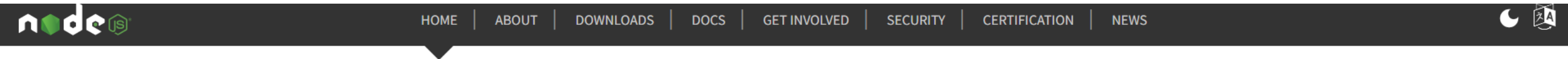


# 0. 개발 환경 설정

Node.js(+npm)

VS Code

# 0.1 Node.js는 무엇인가?



Node.js® is an open-source, cross-platform JavaScript runtime environment.

Node.js assessment of OpenSSL 3.0.7 security advisory

JavaScript runtime environment(자바스크립트 실행 환경)

말그대로 자바스크립트를 실행하기 위한 환경을 제공(가상머신)  
→ 브라우저가 아닌 환경에서도 실행 가능 + 서버 개발 등 다양한 애플리케이션 개발 가능

## 0.2 npm(node package manager)

- Node.js 패키지 매니저
- 프로젝트에서 필요로 하는 다양한 외부 패키지들의 버전과 의존성을 관리
- 편하게 설치 및 삭제할 수 있게 도와주는 역할
- Node.js 설치 시 자동으로 설치됨

## 0.3 Node.js 설치하기

- Node.js 및 npm 버전 확인

 관리자: C:\Windows\system32\cmd.exe

```
C:\Users\gonikim>node -v
v18.12.1

C:\Users\gonikim>node --version
v18.12.1

C:\Users\gonikim>npm -v
7.21.0

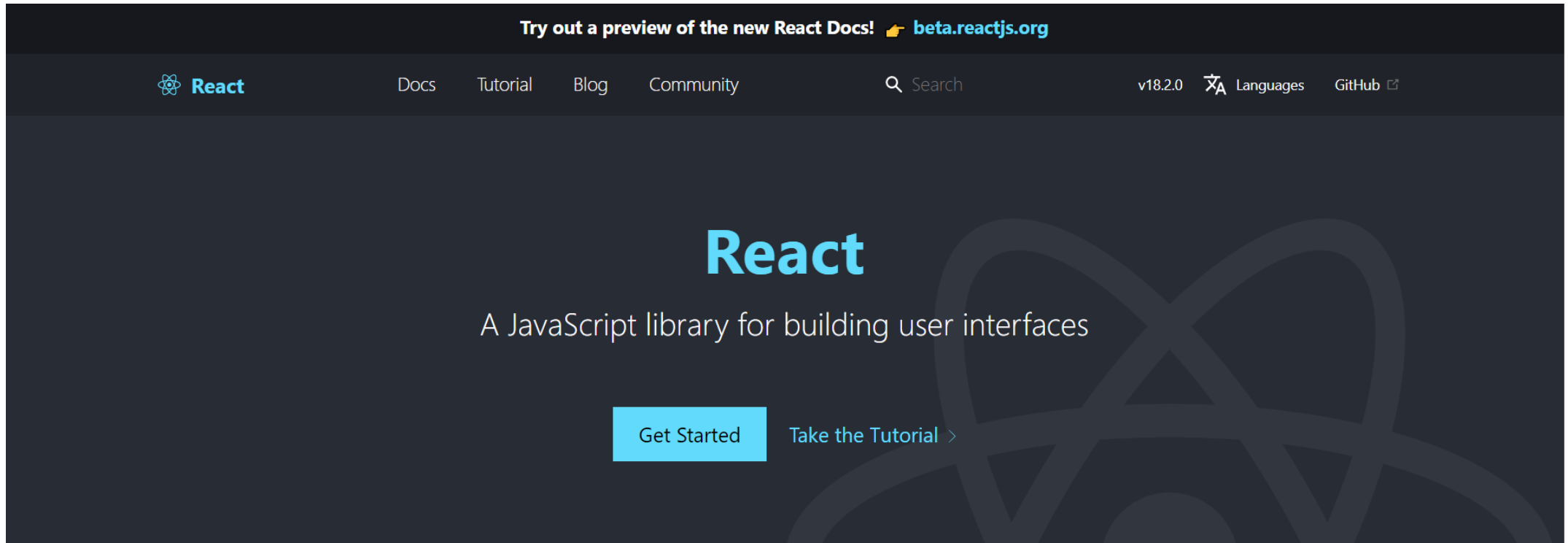
C:\Users\gonikim>npm --version
7.21.0

C:\Users\gonikim>
```

# 1. React 소개

모던하고 트렌디하게 프론트엔드 개발하기

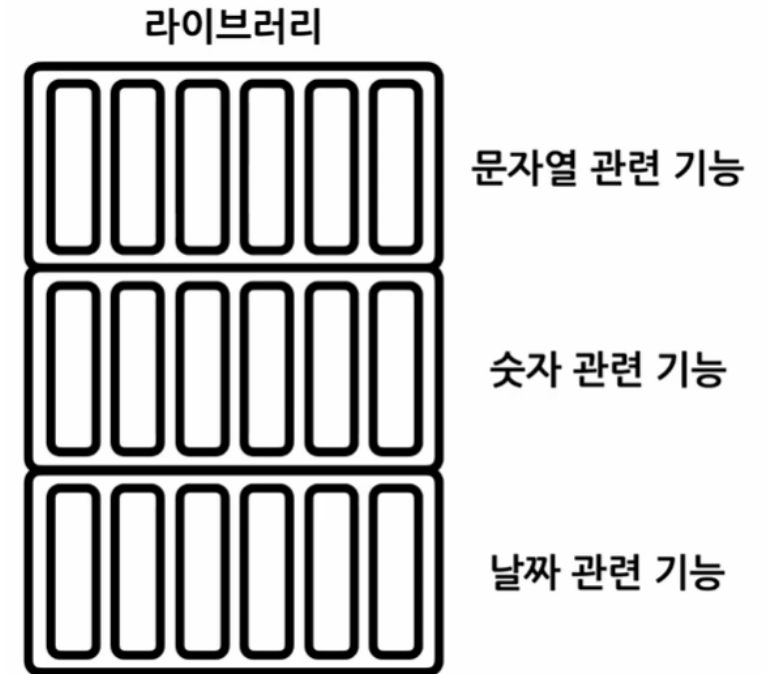
# 1.1 React는 무엇인가?



사용자 인터페이스를 만들기 위한 JavaScript 라이브러리

# 라이브러리? 도서관?

- 도서관에 가면 그림처럼 책장에 수많은 책이 정해진 기준에 따라서 잘 정리되어 있는 것을 볼 수 있음
- 특정 프로그래밍 언어에서 자주 사용되는 기능들을 모아서 정리해 놓은 모음집



# 사용자 인터페이스(User Interface, UI)

- 사용자와 프로그램이 서로 상호작용을 하기 위해 중간에서 입력과 출력을 제어해주는 것
- 웹 사이트를 예로 들면 버튼이랑 텍스트 입력창 등



# React =

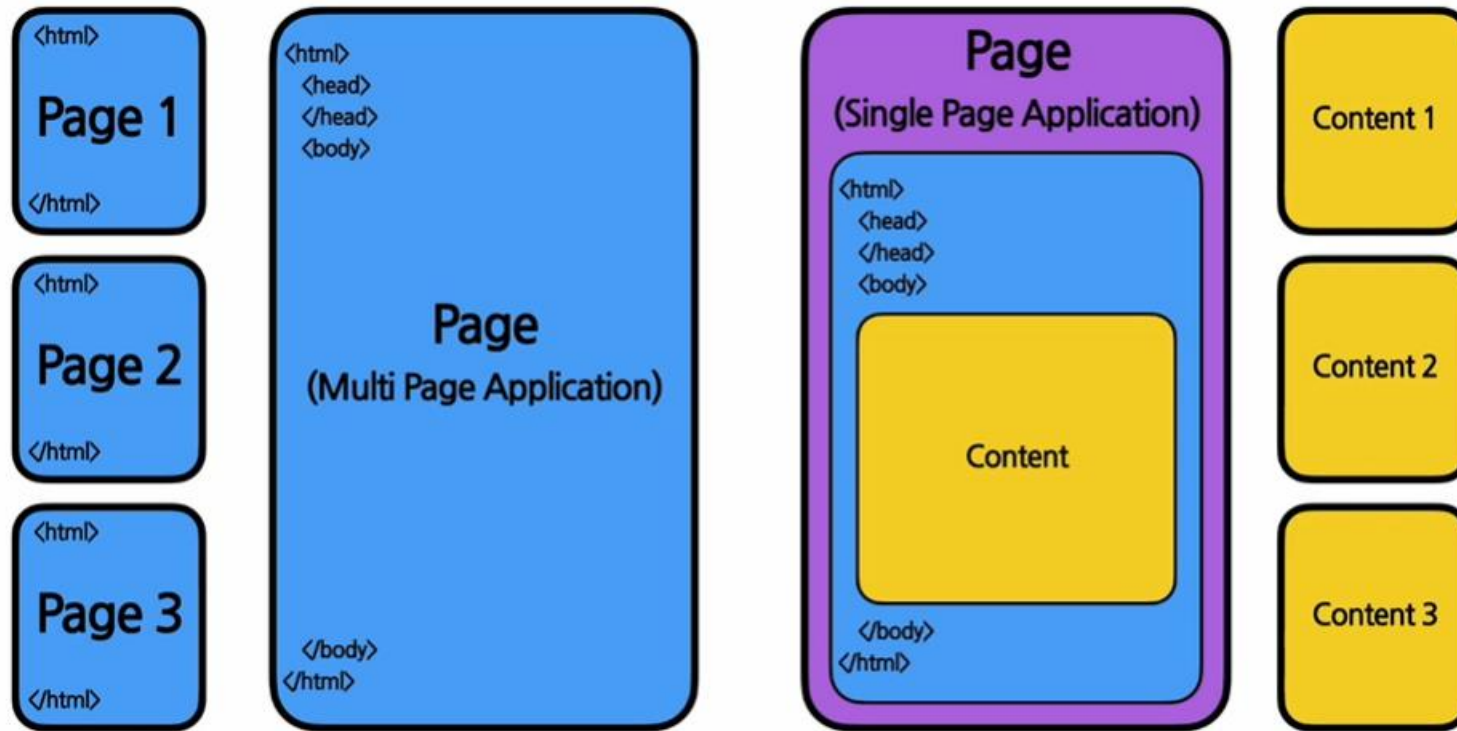
사용자와 웹사이트의 상호작용을 돕는 인터페이스(UI)를  
만들기 위한 자바스크립트 기능 모음집

쉽게 설명하면 화면을 만들기 위한 기능들을 모아 놓은 것  
대표적인 자바스크립트 UI 라이브러리(그 외 Vue.js, Angular)  
SPA 를 쉽고 빠르게 만들 수 있도록 해주는 도구

# SPA(Single Page Application)

- 하나의 페이지만 존재하는 웹 사이트(웹 애플리케이션)을 의미
- 규모가 큰 웹 사이트의 경우 수십~수백 개의 페이지가 존재  
페이지마다 HTML 로 만드는 것은 굉장히 비효율적이고 관리  
하기도 힘들
- 그래서 HTML 틀을 만들어 놓고 사용자가 특정 페이지를 요청  
할 때 그 안에 해당 페이지의 내용만 채우는 것이 바로 SPA
- 이러한 복잡한 웹 사이트를 SPA 방식으로 쉽고 빠르게 만들며  
관리하기 위해 만들어진 것이 바로 React

# MPA vs. SPA



- 각 페이지 별로 HTML 파일이 따로 존재, 페이지를 이동할 때 해당 페이지의 HTML 파일을 받아와서 표시(새로고침)
- 단 하나의 페이지만 존재, 페이지를 이동할 때 해당하는 콘텐츠를 가져와서 동적으로 채워 넣음

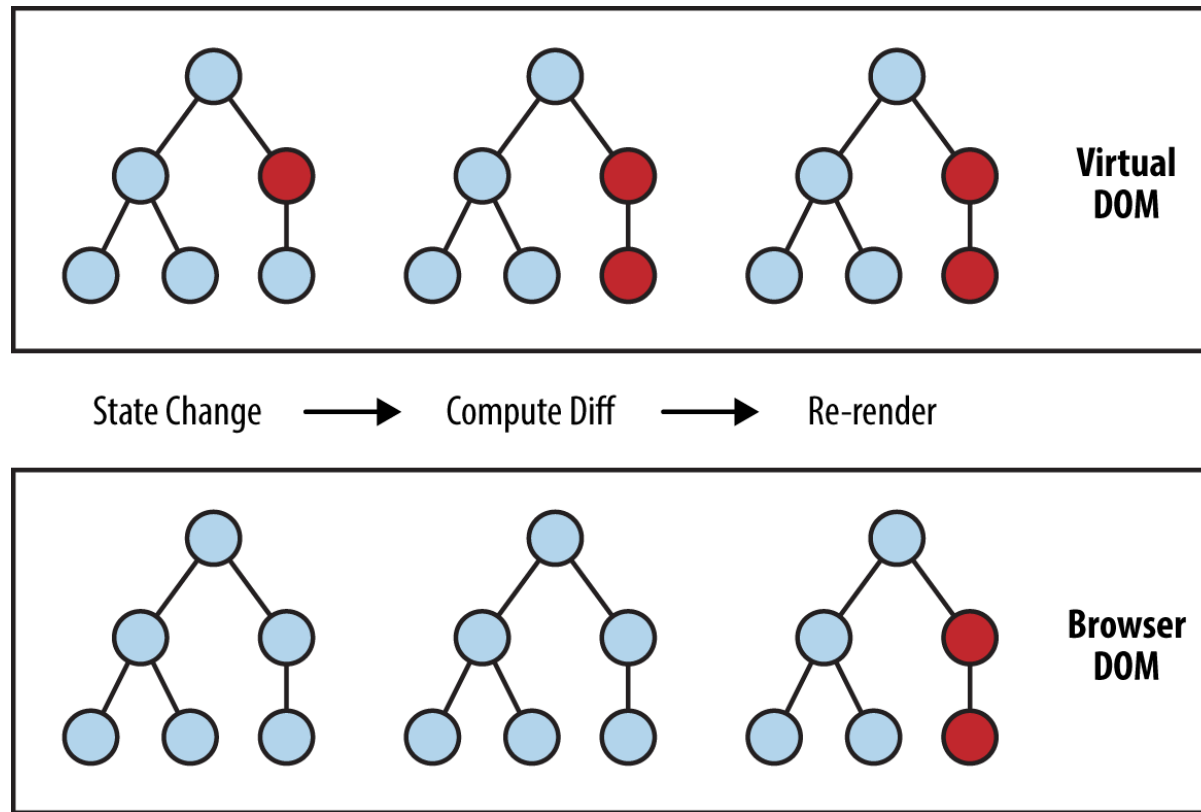
## 1.2 React의 장점

### 1) 빠른 업데이트 & 렌더링 속도

- 여기서 업데이트는 화면에 나타나는 내용이 바뀌는 것을 의미
- 이를 위해 내부적으로 Virtual DOM(가상 돔)을 사용
  - Real DOM(실제 돔)의 사본(가벼움, 빠름)
  - 웹 페이지와 Real DOM 사이에서 중간 매개체 역할
- 기존의 방식: 화면을 업데이트하려면 DOM을 직접 수정 → 성능에 영향(수정할 부분을 DOM에서 모두 찾고 여러 번 연산)
- React 방식: DOM을 직접 수정하는 것이 아니라 가상 돔에 먼저 렌더링하고 이전 가상 돔과 비교하여 업데이트 해야할 최소한의 부분만 찾아서 수정(한번의 연산)

## 1.2 React의 장점

- 해당 엘리먼트와 그 자식 엘리먼트를 이전의 엘리먼트와 비교하고 필요한 경우에만 DOM을 업데이트



# 1.2 React의 장점

## 2) 컴포넌트 구조

- React에서 페이지는 컴포넌트로 구성
- 하나의 컴포넌트는 다른 여러 개의 컴포넌트의 조합으로 구성될 수 있음
- 마치 레고 블록을 조립하여 하나의 모형을 만드는 것과 같음
- 컴포넌트를 조합하여 하나의 웹 사이트를 개발
- 예) 페이스북, 인스타그램, 에어비앤비, 넷플릭스, 트위터 등
- 장점: 컴포넌트의 **재사용** 가능!!

## 1.2 React의 장점

### 3) 재사용성(다시 사용이 가능한 성질)

- 개발 기간이 단축됨
  - 기존에 개발해 둔 모듈을 재사용
- 유지 보수가 용이함
  - 공통으로 사용하는 모듈에 문제가 생기면 해당 모듈만 수정
- React 컴포넌트를 개발할 때 항상 쉽고 재사용 가능한 형태로 개발하는 것이 중요

## 1.2 React의 장점

### 4) 거대한 생태계와 커뮤니티

- 메타(페이스북)의 지원 = 지속적인 업데이트
- 활발한 지식 공유와 질의 응답
  - stack overflow에 질문 수와 watcher 수
- 다양한 라이브러리(for React)
- 오픈소스 라이브러리 인기 지표
  - github 페이지 star수
  - npm trends 다운로드 수



## 1.2 React의 장점

### 5) 모바일 앱 개발 가능

- React 네이티브
  - React를 알면 쉽게 공부 가능
- 하나의 소스로 안드로이드 앱과 iOS 앱을 동시 출시
- 네이티브 앱 보다 성능, 속도 면에서 조금 부족

# 1.3 React의 단점

## 1) 러닝 커브(진입 장벽)

- 방대한 학습량
- 기존과 다른 철학
  - DOM를 직접 제어하던 방식 → 상태 변경
- 빠른 변화(업데이트)
  - 지속적인 공부 필요

# 1.3 React의 단점

## 2) 복잡한 상태 관리

- state라는 중요한 개념!!
- React 컴포넌트의 상태를 의미
- Virtual DOM의 **바뀐 부분**만 찾아서 일괄 업데이트 할 때
  - 바뀐 부분 = state가 바뀐 컴포넌트
- 규모가 큰 앱일수록 높은 상태 관리 복잡도
  - 자체 Context API 사용
  - Redux, MobX, Recoil 등 외부 상태 관리 라이브러리 사용

## 2. React 시작하기(실습)

웹 사이트에 React 추가하기  
새로운 React 앱 만들기(w/CRA)

## 2.1 웹 사이트에 React 추가하기

- React는 처음부터 점진적으로 도입할 수 있게 설계되어서 **필요한 만큼만 사용하면 됨**
- React를 써서 웹 사이트를 만들 때마다 매번 환경 설정을 해주어야 한다면 매우 번거로움
- 새로운 웹 사이트를 만들 때는 처음부터 React 개발 환경이 적용된 상태로 개발을 시작하면 됨

## 2.2 새로운 React 앱 만들기

- React 개발 환경이 적용된 React 프로젝트를 자동으로 생성해주는 Create React App(CRA) 이라는 패키지를 사용할 예정!
- CRA는 npx 명령어를 이용해서 실행
- npx(eXecute NPm package binaries)  
: npm 패키지를 설치하고 실행
- 사용법: `npx create-react-app <your-project-name>`
- 실습: `npx create-react-app my-app`
  - 초기 한 번 CRA 패키지 설치하라고 나오면 y 입력

## 2.2 새로운 React 앱 만들기

- 프로젝트 생성이 완료되면 앱을 실행하기 위한 명령어를 안내
- 경로 변경 (Change Directory)  
`cd my-app`
- 앱 실행  
`npm start`
- localhost:3000 번 포트에서 실행됨

# 3. JSX 소개

React 개발의 필수 이거 안 쓰면 개발 못함



## 3.1 JSX란?

- A syntax extension to JavaScript
- 자바스크립트의 확장 문법
  - JavaScript + XML/HTML
- 간단한 예
  - JS코드(왼쪽)에 HTML코드(오른쪽)가 결합된 모습

```
const element = <h1>Hello, world</h1>;
```

## 3.2 JSX의 역할

- JSX는 내부적으로 XML/HTML 코드를 JS코드로 변환하는 과정을 거침
- 우리가 JSX로 작성을 해도 최종적으로는 JS코드로 변환됨
- JS로 변환된 코드를 JS 객체로 반환시키는 함수
  - `React.createElement()`

## 3.2 JSX의 역할

예를 들어 다음의 JSX로 작성된 코드는

```
class Hello extends React.Component {  
  render() {  
    return <div>Hello {this.props.toWhat}</div>;  
  }  
}  
  
const root = ReactDOM.createRoot(document.getElementById('root'));  
root.render(<Hello toWhat="World" />);
```

아래처럼 JSX를 사용하지 않은 코드로 컴파일될 수 있습니다.

```
class Hello extends React.Component {  
  render() {  
    return React.createElement('div', null, `Hello ${this.props.toWhat}`);  
  }  
}  
  
const root = ReactDOM.createRoot(document.getElementById('root'));  
root.render(React.createElement(Hello, {toWhat: 'World'}, null));
```

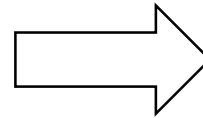
## 3.2 JSX의 역할

- JSX를 사용한 코드(내부적으로는 밑에 코드처럼 변환됨)

```
const element = (  
  <h1 className="greeting">  
    Hello, world!  
  </h1>  
);
```

- JSX를 사용하지 않은 코드(JS코드)

```
const element = React.createElement(  
  'h1', // type: HTML 태그 또는 리액트 컴포넌트  
  { className: 'greeting' }, // [props]: 속성들  
  'Hello, world!' // [...children]: 자식 엘리먼트들  
);
```



- React element(일반 객체)

```
const element = {  
  type: 'h1',  
  props: {  
    className: 'greeting',  
    children: 'Hello, world!'  
  }  
};
```

## 3.3 JSX의 장점

### 1) 코드가 간결함

- JSX 사용 시:
  - `<div>Hello, {name}</div>`
- JSX 미사용 시(only JS):
  - `React.createElement('div', null, `Hello, ${name}`);`

### 2) 가독성이 좋음

- 버그를 발견하기 쉬움!

## 3.3 JSX의 장점

3) Injection Attacks(XSS) 라 불리는 해킹 방법을 방어

- 보안성이 올라감
- 입력창에 들어가는 문자나 숫자 값 대신 소스코드를 주입하여 그 코드가 실행되도록 만드는 해킹 방법
- ReactDOM은 렌더링하기 전에 JSX내에 포함된 모든 값을 문자열로 바꾸기 때문에 공격을 막을 수 있음

## 3.4 JSX의 사용법(예제)

- 기본적으로 JS + HTML 코드
  - ... HTML 코드 ... {JS 코드} ... HTML 코드 ...
  - JSX에서는 중괄호를 사용하면 무조건 JS 코드가 들어감
- my-app/src/chapter3/3.4 에서 실습
  - JsxUse.jsx
  - JsxUse.css

## 3.5 JSX 코드 작성해 보기(실습)

- my-app/src/chapter3 실습
  - Book.jsx
  - Library.jsx



## 4. 엘리먼트 렌더링

화면에 표시할 내용(View단)  
개념만 가볍게 읽고 넘어가기

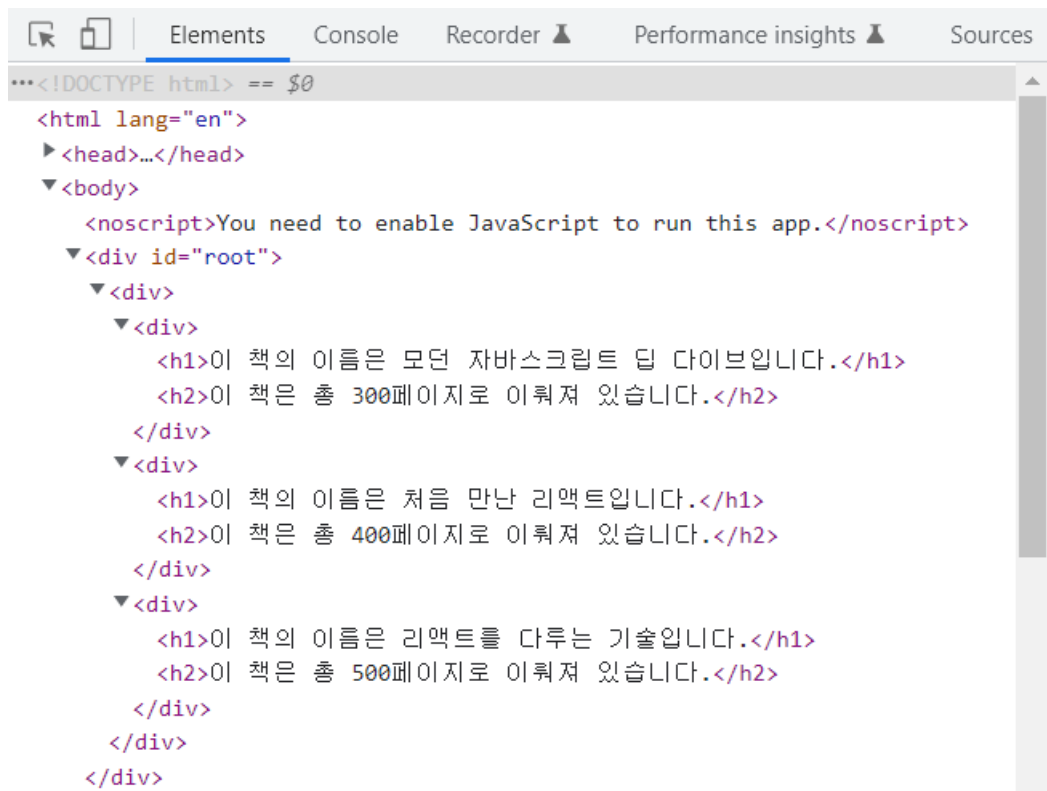
## 4.1 엘리먼트(Elements)란?

- Elements are the smallest building blocks of React apps.
- 엘리먼트는 React 앱을 구성하는 가장 작은 블록들
- 컴포넌트의 "구성 요소"
- 화면에 표시할 내용을 기술하는 자바스크립트 객체
  - 컴포넌트 유형과 속성, 내부의 모든 자식 정보를 포함하는 일반 객체
- 한 번 생성되면 변경할 수 없는 객체(불변성, immutable)
  - 엄밀히 말하면 변경하면 안 되는 객체

```
const element = <h1>Hello, world</h1>;
```

# 4.1 엘리먼트(Elements)란?

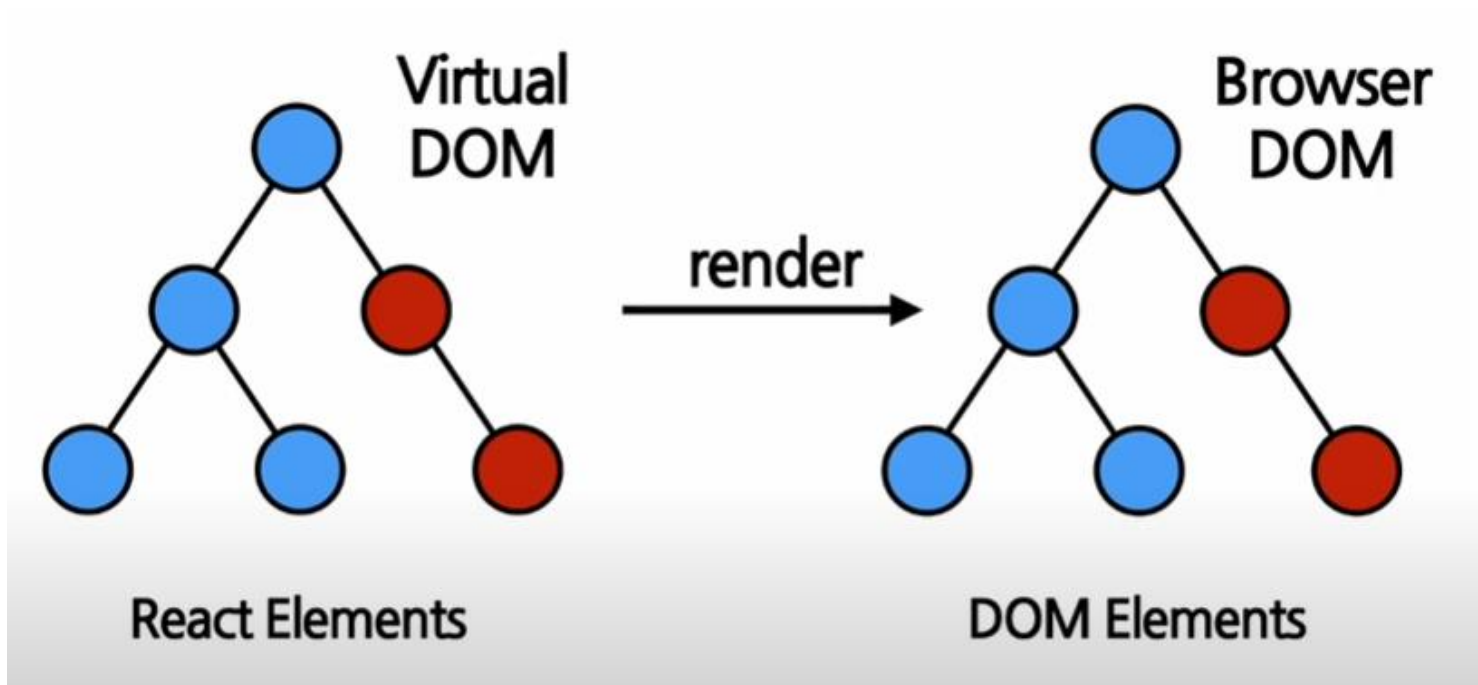
- 우리가 알고 있는 DOM Elements



```
...<!DOCTYPE html> == $0
<html lang="en">
  <head>...</head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root">
      <div>
        <div>
          <h1>이 책의 이름은 모던 자바스크립트 딥 다이브입니다.</h1>
          <h2>이 책은 총 300페이지로 이뤄져 있습니다.</h2>
        </div>
        <div>
          <h1>이 책의 이름은 처음 만난 리액트입니다.</h1>
          <h2>이 책은 총 400페이지로 이뤄져 있습니다.</h2>
        </div>
        <div>
          <h1>이 책의 이름은 리액트를 다루는 기술입니다.</h1>
          <h2>이 책은 총 500페이지로 이뤄져 있습니다.</h2>
        </div>
      </div>
    </div>
  </body>
</html>
```

## 4.1 엘리먼트(Elements)란?

- React 엘리먼트는 DOM 엘리먼트의 가상 표현



## 4.2 엘리먼트의 특징

- 불변성(immutable) = 변하지 않는 성질
  - 한번 생성된 엘리먼트는 변하지 않음
  - 즉, **엘리먼트 생성 후**에는 자식이나 속성을 바꿀 수 없음
  - 예시: 붕어빵 틀(컴포넌트) – 붕어빵(엘리먼트)
- 엘리먼트가 변할 수 없다면 화면 갱신이 안되는 것 아닌가..?
  - 기존 엘리먼트를 변경하는 것이 아니라 새로운 엘리먼트를 만드는 것
  - 새로운 엘리먼트를 기존 엘리먼트와 바꿔치기

## 4.3 엘리먼트 렌더링하기

- root div 안에 들어가는 모든 엘리먼트를 React DOM에서 관리하기 때문에 이것을 루트(Root) DOM 노드라고 부름
- React로 구현된 애플리케이션은 일반적으로 하나의 루트 DOM 노드가 있음

```
<div id="root"></div>
```

```
const root = ReactDOM.createRoot(  
  document.getElementById('root')  
);  
const element = <h1>Hello, world</h1>;  
root.render(element);
```

- root.render(): React 엘리먼트를 DOM 엘리먼트로 렌더링하는 역할
- React 엘리먼트가 렌더링되는 과정 = 가상 DOM에서 실제 DOM으로 이동하는 과정

## 4.4 렌더링 된 엘리먼트 업데이트하기

- React 엘리먼트는 불변 객체
- 엘리먼트를 생성한 이후에는 해당 엘리먼트의 자식이나 속성을 변경할 수 없음
- UI를 업데이트하는 유일한 방법은 새로운 엘리먼트를 생성하고 이를 `root.render()`로 전달

## 4.5 시계 만들기(실습)

- my-app/src/chapter4 실습
  - Clock.jsx
  - 전체 UI를 다시 렌더링 하도록 만들었지만 React DOM은 내용이 변경된 텍스트 노드만 업데이트



# 5. Component와 Props

Component = 레고 블록

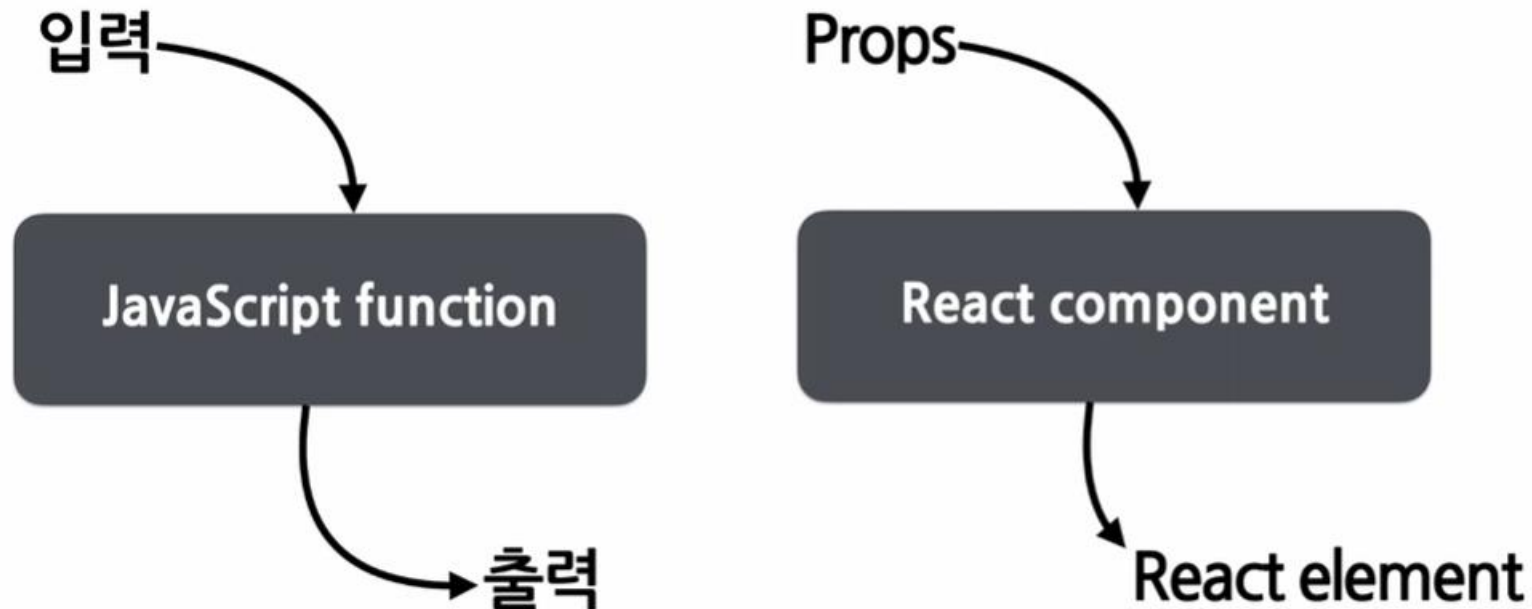
Props = 부모가 자식한테 (객체 형태로) 전달하는 값

## 5.1 컴포넌트(Component)란?

- React는 컴포넌트 기반의 구조
  - 모든 페이지가 컴포넌트로 구성되어 있고, 하나의 컴포넌트는 또 다른 여러 컴포넌트들의 조합으로 만들 수 있음
  - 레고 블록 조립하듯 컴포넌트들을 모아서 개발
  - 반복되는 부분을 컴포넌트로 만들어 재사용
  - 결국 재사용 가능하도록 얼마나 컴포넌트를 잘 쪼개고 잘 조립하느냐가 React 개발

## 5.1 컴포넌트(Component)란?

- 자바스크립트 함수와 유사함
  - Props를 입력으로 받아서 그에 맞는 React 엘리먼트를 생성하여 반환



## 5.2 Props란?

- Property의 줄임으로 React 컴포넌트의 속성들을 의미
  - React 컴포넌트가 엘리먼트를 생성하기 위해 사용하는 값
  - 예시: 붕어빵 틀(컴포넌트) → 재료: 팥, 슈크림 등(Props) → 팥 붕어빵, 슈크림 붕어빵(엘리먼트)
  - 교재 p.145 그림 참고
  - **부모(상위) 컴포넌트가 자식(하위) 컴포넌트에게 값을 전달할 때 사용**
- 컴포넌트에 전달할 다양한 정보를 담고있는 자바스크립트 객체

## 5.3 Props의 특징

- Read-Only
  - 읽기 전용 = 값을 변경 할 수 없다.
  - Props를 전달 받아 엘리먼트가 생성되는 도중에 변경되면 안됨
    - 변경하려고 하면 TypeError 발생
  - 다른 Props의 값으로 엘리먼트를 생성하려면..?
    - 새로운 값을 컴포넌트에 전달하여 새로 엘리먼트를 생성

# [참고] 순수 함수의 개념

- 입력값(input)을 바꾸려 하지 않고 항상 동일한 입력값에 대해 동일한 결과(output)를 반환 = 'Pure'

```
function sum(a, b) {  
  return a + b;  
}
```

- 반면 다음 함수는 자신의 입력값(input)을 변경하기 때문에 순수 함수가 아님 = 'Impure'

```
function withdraw(account, amount) {  
  account.total -= amount;  
}
```

## 5.3 Props의 특징

- 모든 React 컴포넌트는 자신의 props를 다룰 때 반드시 순수 함수처럼 동작해야 한다.
- 모든 React 컴포넌트는 Props를 직접 바꿀 수 없고, 같은 Props에 대해서는 항상 같은 결과(엘리먼트)를 보여줄 것!
- 물론 애플리케이션 UI는 동적으로 변하기 때문에 뒤에서 배울 state를 통해 위 규칙을 위반하지 않고 사용자 액션이나 네트워크 응답 등으로 자신의 출력값(엘리먼트)을 변경 가능

## 5.3 Props의 사용법(예제)

- my-app/src/chapter5/5.3 실습
  - PropsUse.jsx
  - Profile.jsx
  - Layout.jsx
  - Header.jsx
  - Footer.jsx



## 5.4 컴포넌트 만들기

- 컴포넌트의 종류
  - 함수 컴포넌트와 클래스 컴포넌트
- 클래스 컴포넌트
  - ES6의 클래스를 사용하여 만들어진 컴포넌트
  - React 초기 버전에서 주로 사용
  - 사용하기 불편함 → 함수 컴포넌트 + 훅(Hook)으로 대체
- 함수 컴포넌트(권장)
  - 자바스크립트 함수 형태로 된 컴포넌트
  - 코드가 간결해지고 사용하기 편함

## 5.4 컴포넌트 만들기

- 1) 함수 컴포넌트 VS 2) 클래스 컴포넌트
  - 1) Welcome이라는 이름의 함수가 props 객체를 전달받아서 인사말이 담긴 React 엘리먼트를 반환

```
function Welcome(props) {  
  return <h1>Hello, {props.name}</h1>;  
}
```

- 2) Welcome이라는 클래스

```
class Welcome extends React.Component {  
  render() {  
    return <h1>Hello, {this.props.name}</h1>;  
  }  
}
```

## 5.4 컴포넌트 만들기

- 컴포넌트의 이름 짓기
  - 항상 대문자로 시작해야 한다!
  - React는 소문자로 시작하는 컴포넌트를 HTML DOM Tag로 인식하려고 하기 때문
  - HTML div 태그로 인식

```
const element = <div />;
```

- Welcome이라는 React 컴포넌트로 인식

```
const element = <Welcome name="Sara" />;
```

## 5.4 컴포넌트 만들기

- 컴포넌트 렌더링 - 페이지에 "Hello, Sara"를 렌더링하는 예시

```
function Welcome(props) {  
  return <h1>Hello, {props.name}</h1>;  
}  
  
const root = ReactDOM.createRoot(document.getElementById('root'));  
const element = <Welcome name="Sara" />;  
root.render(element);
```

- React는 { name: 'Sara' }를 props로 하여 Welcome 컴포넌트를 호출
- Welcome 컴포넌트는 <h1>Hello, Sara</h1> 엘리먼트를 반환
- 엘리먼트를 인자값으로 root.render()를 호출
- React DOM은 <h1>Hello, Sara</h1> 엘리먼트를 실제 DOM에 효율적으로 업데이트

## 5.5 컴포넌트 합성

- 여러 개의 컴포넌트를 합쳐서 하나의 컴포넌트를 만드는 것
- 컴포넌트 안에 또 다른 컴포넌트를 쓸 수 있음
- 복잡한 화면을 여러 개의 컴포넌트로 나눠서 구현 가능!

## 5.5 컴포넌트 합성

- 예시: Welcome 을 여러 번 렌더링하는 App 컴포넌트
  - App 컴포넌트는 3개의 Welcome 컴포넌트를 포함
  - 이렇게 여러 개의 컴포넌트를 합쳐서 또 다른 컴포넌트를 만드는 것을 컴포넌트 합성이라고 함

```
function Welcome(props) {  
  return <h1>Hello, {props.name}</h1>;  
}  
  
function App() {  
  return (  
    <div>  
      <Welcome name="Sara" />  
      <Welcome name="Cahal" />  
      <Welcome name="Edite" />  
    </div>  
  );  
}
```

## 5.6 컴포넌트 추출

- 큰 컴포넌트에서 일부를 추출해서 새로운 컴포넌트를 만드는 것
- 복잡한 컴포넌트를 쪼개서 나누는 작업
- 컴포넌트를 어느 정도 수준까지 추출하는 것이 좋은지에 대해 정해진 기준은 없음
- 기능 단위로 구분하는 것이 좋고, 나중에 곧바로 **재사용이 가능**한 형태로 추출하는 것이 좋음
- 재사용 가능한 컴포넌트를 많이 갖고 있을수록 개발 속도가 빨라짐

## 5.6 컴포넌트 추출(예제)

- my-app/src/chapter5/5.6 실습
  - Comment.jsx
  - Avatar.jsx
  - UserInfo.jsx



## 5.7 댓글 컴포넌트 만들기(실습)

- my-app/src/chapter5 실습
  - Comment.jsx
  - CommentList.jsx
  - index.js

## 6. State와 Lifecycle

State(매우 중요) = 화면이 바뀌어야 할 때 state를 변경  
Lifecycle(클래스 컴포넌트와 밀접, 개념만 슬쩍~)

## 6.1 State란?

- React 컴포넌트의 상태  
→ React 컴포넌트의 “변경 가능한” 데이터를 의미
- **컴포넌트에서 보여줘야 하는 내용이 사용자 인터랙션에 따라 동적으로 바뀌어야 할 때 사용**
- 렌더링과 관련된 값만 state에 포함시켜야 함
  - 이유? state가 변경될 경우 컴포넌트가 재렌더링 되기 때문
- 렌더링과 관련 없는 값을 포함하면 불필요한 재렌더링 발생으로 성능이 저하될 수 있음

## 6.2 State의 특징

- 자바스크립트 객체 형태로 존재(in 클래스 컴포넌트)
- 일반적인 객체의 값을 수정하듯이 **직접적으로 변경하면 안됨**
  - 값은 변경되지만 재렌더링(화면 업데이트)이 안됨
- ~~클래스 컴포넌트~~
  - 생성자에서 모든 state를 한 번에 정의
  - state를 변경하고자 할 때에는 꼭 **setState()** 함수를 사용
- 함수 컴포넌트
  - useState() Hook을 사용하여 각각의 state를 정의
  - 각 state별로 주어지는 **set함수를 사용하여** state 값을 **변경**

## 6.3 Lifecycle

- React 컴포넌트의 생명주기
  - 예: 컴포넌트가 생성되는 시점과 사라지는 시점 등
- 컴포넌트는 계속 존재하는 것이 아니라 데이터의 흐름에 따라 생성되고 업데이트 되다가 사라지는 과정을 겪음
- 참고: <https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/>
- 생명주기는 이런 개념이 있다 정도로만 이해하고 넘어가기!

## 6.3 Lifecycle

- 마운트(Mounting)
  - 컴포넌트가 생성될 때(화면에 나타날 때)
  - constructor() 호출
  - render() 호출
  - componentDidMount() 호출

## 6.3 Lifecycle

- 업데이트(Updating)
  - 컴포넌트가 업데이트 될 때(화면이 바뀔 때)  
= 바뀐 부분에 대한 재렌더링이 일어날 때
  - **업데이트가 일어나는 조건?**
    - New props: 컴포넌트에 새로운 props가 전달될 때
    - setState() 함수 호출에 의해 state가 변경될 때
    - forceUpdate()라는 강제 업데이트 함수가 호출될 때
  - render() 호출
  - componentDidUpdate() 호출

## 6.3 Lifecycle

- 언마운트(Unmounting)
  - 컴포넌트가 제거될 때(화면에서 사라질 때)
    - (상위 컴포넌트에서) 현재 컴포넌트를 더 이상 화면에 표시하지 않게 될 때
  - 언마운트 직전에 `componentWillUnmount()` 호출
- ~~이 외에도 생명주기 메서드가 더 존재하지만...~~



## 6.4 state 사용하기(실습)

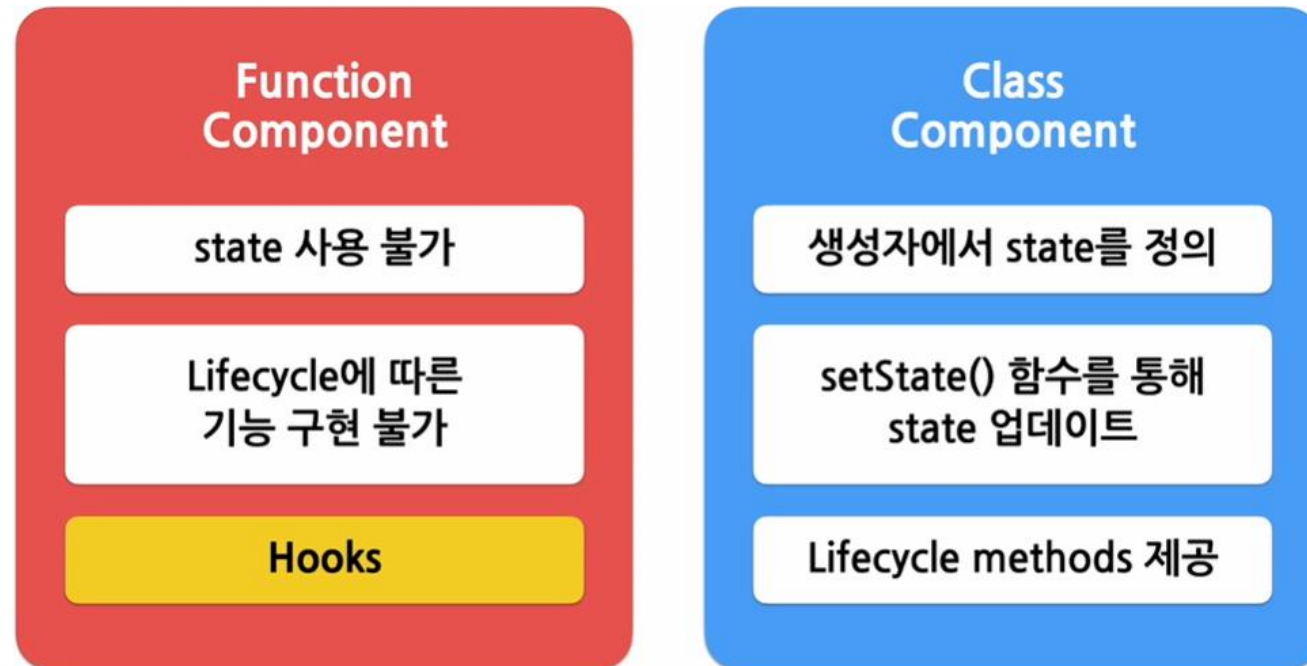
- my-app/src/chapter6 실습
- 1) state 사용하기
- 2) React Developer Tools 설치 및 사용하기
  - React를 위해 별도로 개발된 React 개발자 도구
  - 구글 검색 후 크롬 웹 스토어에서 설치
  - 크롬 개발자 도구에 Components와 Profiler 탭이 생김
  - 컴포넌트를 누르면 props와 state 확인 가능
- 3) 생명주기 메서드 사용해보기

# 7. Hook

클래스 bye, 지금은 함수의 시대

## 7.1 Hook이란?

- Hook은 React 버전 16.8부터 새롭게 추가



- **Hook은 Class없이 React 기능들을 사용하는 방법을 제시**
  - 대표적인 기본 Hook → useState(), useEffect()

## 7.1 Hook이란?

- 원래 존재하는 어떤 기능에 마치 갈고리를 거는 것처럼 끼어 들어 수행되는 것
- Hook은 함수 컴포넌트에서 React state와 생명주기 기능 (lifecycle features)을 "연동(hook into)"할 수 있게 해주는 함수
- Hook은 class 안에서는 동작하지 않고 대신 class 없이 React를 사용할 수 있게 해주는 것
- 이 함수들을 Hook이라 부르고 이름은 모두 use로 시작함

## 7.2 useState(예제)

- 가장 대표적이고 많이 사용되는 Hook
- **state를 생성하고 변경**하기 위한 Hook
- 함수 컴포넌트에서는 기본적으로 state라는 것을 제공하지 않음
- 클래스 컴포넌트처럼 state를 사용하고 싶으면 useState()를 사용해야 함
- 사용법
  - `const [변수명, set함수명] = useState(초기값);`
  - 변수 각각에 대해 set 함수가 따로 존재함

## 7.3 useEffect

- useState와 같이 가장 많이 사용되는 Hook
- ~~사이드 이펙트를 수행하기 위한 Hook~~
  - 사이드 이펙트: 함수의 핵심 기능과 상관없는 부가 기능  
(예: 서버에서 데이터를 받아오거나 수동으로 DOM을 변경하는 등의 작업)
  - 컴포넌트가 렌더링 이후에 어떤 일을 수행해야하는 지를 말함  
(effect가 수행되는 시점에 이미 DOM이 업데이트 되었음을 보장)
- 마운트/업데이트/언마운트 시 할 작업 설정
  - 컴포넌트가 마운트 됐을 때 (처음 나타났을 때)
  - 업데이트(재렌더링) 될 때 (props 또는 state가 바뀔 때)
  - 언마운트 됐을 때 (사라질 때) 특정 작업을 처리

## 7.3 useEffect

- useEffect() 만으로 클래스 컴포넌트의 생명주기 메서드들과 동일한 기능을 수행할 수 있음
  - componentDidMount()
  - componentDidUpdate()
  - componentWillUnmount()
  - 위 3개를 하나로 통합해서 제공

## 7.3 useEffect(예제)

- 사용법

- useEffect(이펙트 함수, 의존성 배열);
- 의존성 배열 생략 시 **매번 렌더링 될 때마다** 실행  
(마운트 + 업데이트 될 때마다 호출됨)
- 의존성 배열 안에 있는 변수들 중 하나라도 **값이 변경되었을 때** 실행됨  
(마운트 + 값 변경에 의한 업데이트 시 호출됨)
- 의존성 배열에 빈 배열([])을 넣으면 **화면에 첫 렌더링 될 때만** 실행  
(마운트)
- useEffect()에서 리턴하는 함수는 언마운트 될 때 호출됨
  - clean-up(정리) 함수



## 7.3 useEffect

- 사용법

```
useEffect(() => {  
  // 컴포넌트가 마운트 된 이후,  
  // 의존성 배열에 있는 변수들 중 하나라도 값이 변경되었을 때 실행됨  
  // 의존성 배열에 빈 배열([])을 넣으면 마운트 시에 단 한 번씩만 실행됨  
  // 의존성 배열 생략 시 컴포넌트 업데이트 시마다 실행됨  
  ...  
  
  return () => {  
    // 컴포넌트가 마운트 해제되기 전에 실행됨  
    ...  
  }  
}, [의존성 변수1, 의존성 변수2, ...]);
```

## 7.4 useMemo

- Memoized value를 리턴하는 Hook
  - Memoization 개념: 연산량이 많은 동일한 계산을 반복해야 할 때, 이전 결과를 메모리에 저장해 두었다가 동일한 계산에서 쓰는 것
  - 즉, 이전에 계산 한 값을 재사용
- 연산량이 높은 작업이 매번 렌더링 될 때마다 반복되는 것을 피하기 위해 사용
- 쉽게 말해 자주 필요한 값을 맨 처음에 저장해뒀다가 필요할 때마다 꺼내 사용
- 최적화와 관련된 Hook

## 7.4 useMemo

- 사용법

- `const 변수명 = useMemo(값 생성 함수, 의존성 배열);`
- 의존성 배열에 들어있는 변수가 변했을 경우에만 새로 값 생성 함수를 호출하여 결과값을 반환함
- 그렇지 않은 경우에는 기존 함수의 결과값을 그대로 반환함
- 의존성 배열을 넣지 않을 경우 렌더링이 일어날 때마다 매번 값 생성 함수가 실행되므로 의미가 없음

## 7.4 useMemo(예제)

- 사용법

```
const memoizedValue = useMemo(  
  () => {  
    // 연산량이 높은 작업을 수행하여 결과를 반환  
    return computeExpensiveValue(의존성 변수1, 의존성 변수2);  
  },  
  [의존성 변수1, 의존성 변수2]  
);
```

## 7.5 useCallback

- useMemo() Hook과 유사하지만 값이 아닌 함수를 반환한다는 점이 다름
- 컴포넌트 내에 함수를 정의하면 매번 렌더링이 일어날 때마다 함수가 새로 정의되므로 useCallback() Hook을 사용하여 불필요한 함수 재정의 작업을 없애는 것
- 최적화와 관련된 Hook
- 사용법
  - `const 함수명 = useCallback(콜백 함수, 의존성 배열);`
  - 의존성 배열에 들어있는 변수가 변했을 경우에만 콜백 함수를 다시 정의해서 리턴함

## 7.5 useCallback(예제)

- 사용법

```
const memoizedCallback = useCallback(  
  () => {  
    doSomething(의존성 변수1, 의존성 변수2);  
  },  
  [의존성 변수1, 의존성 변수2]  
);
```

## 7.6 useRef(예제 1)

- 레퍼런스를 사용하기 위한 Hook
  - 레퍼런스(참조)란 특정 컴포넌트에 접근할 수 있는 객체
- 매번 렌더링 될 때마다 **항상 같은 레퍼런스 객체**를 반환
- 반환된 객체는 **컴포넌트의 전 생애주기에 걸쳐서 유지됨**
  - 컴포넌트가 마운트 해제 되기 전까지 계속 유지
- 부득이하게 특정 DOM을 선택해야 하는 경우에만 사용
  - In JavaScript, getElementById(), querySelector() 같은 DOM Selector 함수를 사용
  - In React, useRef() Hook 사용
  - 예를 들어 특정 엘리먼트의 크기가 필요할 때, 스크롤바 위치를 가져오거나 설정해야 될 때, 또는 input 같은 상호작용 가능한 엘리먼트에 포커스를 줘야 할 때 등

## 7.6 useRef(예제2)

- 또 다른 용도로 컴포넌트 안에 변수 만들 때 사용
  - 컴포넌트 안에서 조회 및 수정 할 수 있는 지역 변수
  - useRef 로 관리하는 변수는 값이 바뀐다고 해서 컴포넌트가 리렌더링 되지 않음
  - 렌더링이 될 때 값이 초기화 되지 않음  
(일반 변수로 선언 시 렌더링이 될 때 마다 값이 초기화 됨)
- 사용법
  - `const 변수명 = useRef(초기값);`
  - `변수명.current`라는 속성을 통해서 접근
    - `current`: 현재 참조하고 있는 엘리먼트(또는 값)
    - `current` 속성에 변경 가능한 값을 담고 있는 "상자"와 같음



## 7.7 Hook의 규칙

- 무조건 최상위 레벨에서만 호출해야 함
  - 함수 컴포넌트 안에서 최상위 레벨을 의미
  - 반복문이나 조건문 또는 중첩된 함수들 안에서 Hook을 호출하면 안됨
    - Hook의 실행 순서가 바뀔 수 있음  
→ state가 꼬이거나 렌더링 문제 유발
- 함수 컴포넌트에서만 Hook을 호출해야 함
  - 또는 직접 만든 커스텀 Hook에서만 호출 가능

```
function HookRules(props) {
  const [name, setName] = useState('Goni');

  useEffect(() => {
    // "Good"
  });

  if (name !== '') {
    useEffect(() => {
      // ... Bad ...
    });
  }

  for (let index = 0; index < 3; index++) {
    useEffect(() => {
      // ... Bad ...
    });
  }

  function func() {
    useEffect(() => {
      // ... Bad ...
    });
  }

  return (
    <div>
    </div>
  )
};
```

## 7.8 Custom Hook 만들기(예제)

- 기본적으로 제공되는 Hook 이외에 추가적으로 필요한 기능이 있다면 직접 만들어 사용
- 이것을 커스텀 훅이라 부름
- 여러 컴포넌트에서 반복적으로 사용되는 로직을 훅으로 만들어 재사용
- 자바스크립트에서 특정 기능을 함수로 만들어 재사용 하는것과 같음

## 7.8 Custom Hook 만들기

- 정리

- 이름이 use로 시작하여 React Hook임을 알리고, 내부에서 다른 Hook을 호출하는 **단순한 자바스크립트 함수**
- 함수를 만들 때 파라미터로 무엇을 받을지, 어떤 것을 리턴 할지는 개발자가 직접 정하면 됨
- **중복되는 로직을 커스텀 훅으로 추출하여 재사용성 높이기**
- 만약 여러 개의 컴포넌트에서 하나의 커스텀 훅을 사용할 때 훅 내부에 있는 state를 공유하는 것일까..?
  - 그렇지 않다.
  - 커스텀 훅을 호출할 때마다 별개의 state를 얻게 됨!

## 7.9 훅을 사용한 컴포넌트 개발(실습)

- my-app/src/chapter7 실습
- 1) useCounter() 커스텀 훅 만들기
- 2) Accommodate 컴포넌트 만들기
- 3) 실행하기

## 8. 이벤트 처리하기

onclick no, onClick ok

이벤트를 처리하는 함수 → 이벤트 핸들러

## 8.1 이벤트 처리 방식(예제)

- React 엘리먼트에서 이벤트를 처리하는 방식은 DOM 엘리먼트에서 이벤트를 처리하는 방식과 매우 유사
  - React의 이벤트는 소문자 대신 카멜 케이스(camelCase)를 사용
  - JSX를 사용하여 문자열이 아닌 함수로 전달

예를 들어, HTML은 다음과 같습니다.

```
<button onclick="activateLasers()">  
  Activate Lasers  
</button>
```

React에서는 약간 다릅니다.

```
<button onClick={activateLasers}>  
  Activate Lasers  
</button>
```

## 8.2 인자 전달하기(예제)

- Arguments: 인수, 인자(값), 매개값, 함수에 전달할 데이터
- Parameter: 매개변수, 전달 받은 데이터를 저장할 변수
- 함수를 하나 만들어서 그 안에서 Arguments를 넣어 호출

## 8.3 클릭 이벤트 처리하기(실습)

- my-app/src/chapter8 실습
  - ConfirmButton
- ~~클래스 컴포넌트 실습~~
- 함수 컴포넌트 실습



## 9. 조건부 렌더링

true이면 버튼을 보여주고, false이면 버튼을 숨기고

## 9.1 조건부 렌더링이란?(예제)

- 조건에 따라 렌더링의 결과가 달라지도록 하는 것
  - 예제: 로그인 여부에 따라 보여줄 인사말이 달라지도록 만들기
- 자바스크립트의 Truthy, Falsy
  - true로 여겨지는 값 → truthy
  - false로 여겨지는 값 → falsy
    - false
    - 0, -0
    - "", "", `` (empty string)
    - null
    - undefined
    - NaN (Not a Number)

## 9.2 엘리먼트 변수(예제)

- 조건부 렌더링 시 컴포넌트를 변수처럼 다루고 싶을 때
- React 엘리먼트를 변수처럼 저장해서 사용하는 방법

## 9.3 인라인 조건(예제)

- **논리 && 연산자로 If를 인라인으로 표현하기**

- return (); 안의 JSX 내에서는 if문 사용 불가
- Short-circuit evaluation(단축 평가)
  - true && expression → expression
  - false && expression → false
- 앞에 나오는 조건이 true일 경우에만 뒤에 나오는 엘리먼트를 렌더링

- **삼항 연산자로 If-Else구문 인라인으로 표현하기**

- 조건에 따라 각기 다른 엘리먼트를 렌더링하고 싶을 때 사용

## 9.4 컴포넌트의 렌더링 막기(예제)

- React에서는 null을 리턴하면 렌더링되지 않음
- 특정 컴포넌트를 렌더링하고 싶지 않을 경우 null을 리턴하면 됨

## 9.5 로그인 여부 톨바 만들기(실습)

- my-app/src/chapter9 실습
  - Toolbar
  - LandingPage

# 10. 리스트와 Key

리스트(목록) = 같은 아이템을 순서대로 모아놓은 것  
리스트를 담는 자료구조 → 배열

Key = 리스트의 아이템들을 구분하기 위한 고유한 값

# 10.1 여러 개의 컴포넌트 렌더링 하기(예제)

- 배열을 사용하여 반복되는 여러 개의 컴포넌트들을 쉽게 렌더링 가능
- 컴포넌트를 코드 상에 하나씩 직접 넣는 것은 비효율적
- 배열의 `map()` 함수를 이용
  - `map()`: 배열의 각 요소에 어떤 처리를 한 뒤 리턴하는 것으로 새로운 배열을 만듦
  - 어떤 처리? 리액트 엘리먼트로 만들기
  - `map()` 함수 안에 있는 엘리먼트에는 무조건 `Key`가 있어야 함



## 10.2 리스트의 Key(예제)

- 리스트에서 아이템들을 구분하기 위한 고유한 문자열
- 어떤 아이템이 추가, 변경, 삭제되었는지 구분하기 위해 사용
  - (참고) 키가 없다면 재렌더링 시 엘리먼트를 비효율적으로 업데이트
- 다양한 키 값의 사용법
  - **데이터의 id를 사용 (주로 사용)**
  - 인덱스를 사용 (데이터에 고유한 id값이 없을 경우에만 사용)
    - 아이템의 순서가 바뀔 수 있는 경우 권장 안함
    - <https://robinpokorny.medium.com/index-as-a-key-is-an-anti-pattern-e0349aece318>

## 10.3 출석부 출력하기(실습)

- my-app/src/chapter10 실습

# 11. Form

사용자로부터 입력을 받기 위해 사용하는 것(입력 양식)  
대표적으로 <input>

# 11.1 HTML Form

- submit하면 지정한 URL로 요청을 보내는 기본 폼 동작을 수행

HTML 폼 엘리먼트는 폼 엘리먼트 자체가 내부 상태를 가지기 때문에, React의 다른 DOM 엘리먼트와 다르게 동작합니다. 예를 들어, 순수한 HTML에서 이 폼은 name을 입력받습니다.

```
<form>
  <label>
    Name:
    <input type="text" name="name" />
  </label>
  <input type="submit" value="Submit" />
</form>
```

- 그러나 대부분의 경우, 자바스크립트 코드에서 사용자가 입력한 데이터에 접근하고 처리 하는 것이 편리
- React에서는 이를 위해 제어 컴포넌트(controlled components)

## 11.2 제어 컴포넌트(예제)

- 사용자가 입력한 값에 접근하고 제어할 수 있도록 해주는 컴포넌트
- 입력값이 state를 통해 리액트의 통제를 받는 입력 폼 엘리먼트

## 11.3 다양한 Form(예제)

- `<textarea>` 태그
  - 여러 줄에 걸쳐서 텍스트를 입력 받기 위한 HTML 태그
  - `value` 속성으로 입력된 값을 관리
- `<select>` 태그
  - 드롭다운 목록을 보여주기 위한 HTML 태그
  - 여러 옵션 중에서 하나 또는 여러 개를 선택할 수 있는 기능을 제공
  - `value` 속성으로 입력된 값을 관리
- `<input type="file">` 태그
  - 하나 또는 여러 개의 파일을 선택할 수 있게 해주는 HTML 태그
  - 서버로 파일을 업로드하거나 자바스크립트로 파일을 다룰 때 사용
  - 값이 읽기 전용이기 때문에 React에서는 비제어 컴포넌트가 됨

## 11.4 여러 개의 입력 제어하기(예제)

- 컴포넌트에 여러 개의 state를 선언하여 각각의 입력에 대해 사용하면 됨

## 11.5 Input Null 값에 대해(참고)

- 값이 비어 있음을 나타낼 때 value 속성에 null값 사용은 안됨
  - 일반적으로 제어 컴포넌트에서는 빈 문자("")를 사용
  - 비제어 컴포넌트에서는 undefined 값을 사용



## 11.6 사용자 정보 입력받기(실습)

- my-app/src/chapter11 실습

# 간단한 포스트 만들기

my-post 실습

## 12. State 끌어올리기

여러 개의 컴포넌트들 사이에서 state를 공유하려면..?  
공통된 부모 컴포넌트로 끌어올려서 공유하는 방식

## 12.1 Shared State

- 종종 동일한 데이터에 대해 여러 컴포넌트에서 필요할 경우 각 컴포넌트의 state에서 데이터를 각각 보관하는 것이 아니라 가장 가까운 공통된 조상 컴포넌트의 state를 공유해서 사용하는 것이 더 효율적
- 즉, 하위 컴포넌트가 공통된 상위 컴포넌트의 state를 공유하여 사용하는 것
- 어떻게 공유..? (뒷장)

## 12.2 State를 끌어올려 공유하기(실습)

- 하위 컴포넌트의 state를 공통된 부모 컴포넌트로 끌어올려서 공유하는 방식
- 그런 다음에 하위 컴포넌트에 props로 state를 넘겨줌
- my-app/src/chapter12 실습

# 13. 합성 vs 상속

상속 대신 합성

# 13.1 합성(Composition)(예제)

- 여러 개의 컴포넌트를 합쳐서 새로운 컴포넌트를 만드는 것
- 컴포넌트들을 어떻게 조합할 것인가?
  - 1) Containment(컴포넌트에서 다른 컴포넌트를 담기)
    - 하위 컴포넌트를 포함하는 형태
    - 어떤 자식 엘리먼트가 들어올 지 미리 예상할 수 없는 경우
    - props에 기본적으로 들어있는 children 속성을 사용
  - 2) Specialization(특수화)
    - 예를 들어, WelcomeDialog(구체적)는 Dialog(범용적)의 특수한 경우이다.
    - 범용적인 개념을 구별이 되게 구체화 하는 것
    - 범용적인 컴포넌트를 만들어 놓고 이를 구체화시켜서 사용
  - 3) 두 개를 같이 사용

## 13.2 상속(Inheritance)

- Facebook에서는 수천 개의 React 컴포넌트를 사용하지만, 컴포넌트를 상속 계층 구조로 작성을 권장할만한 사례를 아직 찾지 못했습니다.
- 결론은 **복잡한 컴포넌트를 쪼개 재사용 가능한 여러 개의 컴포넌트로 만들고, 만든 컴포넌트들을 조합하여 새로운 컴포넌트를 만들자!**



## 13.3 Card 컴포넌트 만들기(실습)

- my-app/src/chapter13 실습

# 14. Context

일일이 props를 넘겨주지 않고도  
컴포넌트 트리 전체에 데이터를 제공 가능

## 14.1 컨텍스트(Context)란?

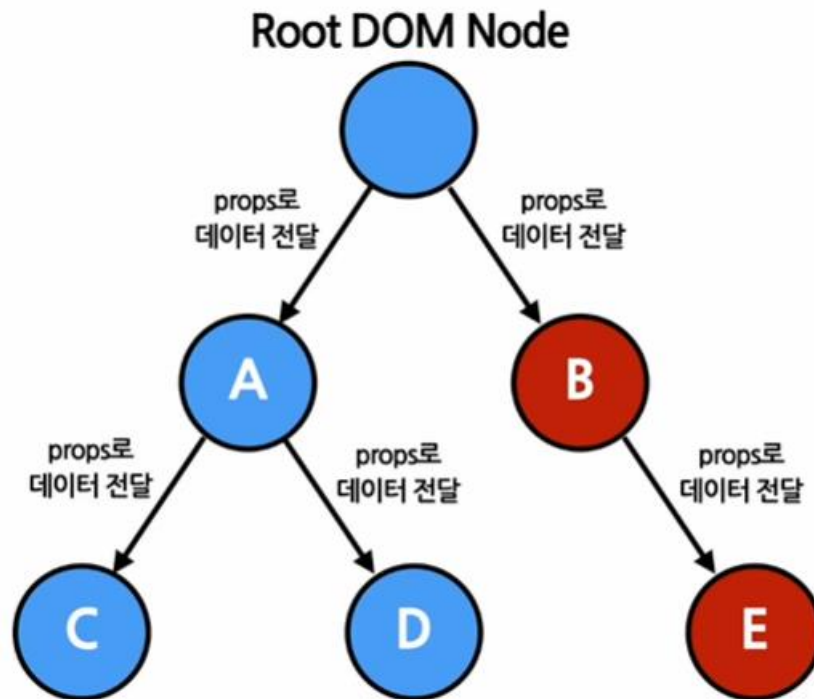
- 컴포넌트들 사이에서 데이터를 props를 통해 전달하는 것이 아닌 컴포넌트 트리를 통해 곧바로 데이터를 전달하는 방식
- 어떤 컴포넌트든지 컨텍스트에 있는 데이터에 쉽게 접근 가능

# 14.1 컨텍스트(Context)란?

- 일반적인 React 앱에서 데이터는 위에서 아래로(부모로부터 자식에게) props를 통해 전달되지만, 앱 안의 여러 컴포넌트들에 전해줘야 하는 props의 경우 이 과정이 번거로울 수 있음

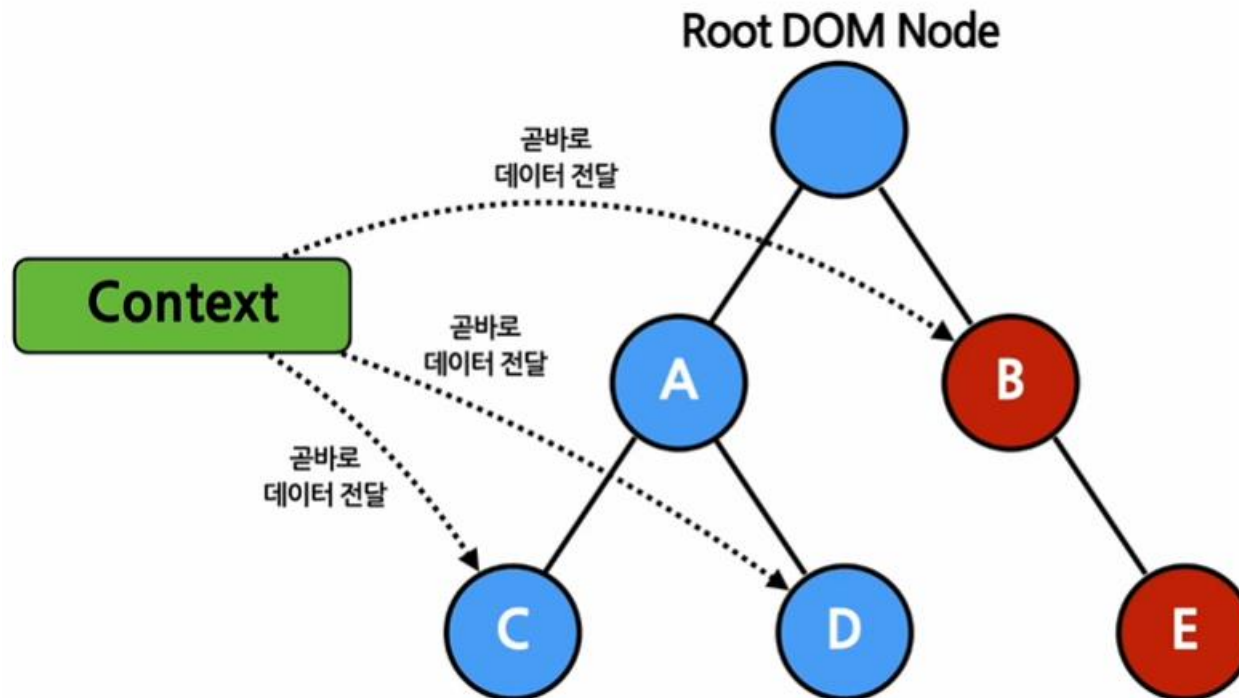
- 예를 들면

- Locale 정보
- UI 테마
- 로그인 유저 정보
- 앱 환경설정



# 14.1 컨텍스트(Context)란?

- context를 이용하면, 트리 단계마다 명시적으로 props를 넘겨주지 않아도 많은 컴포넌트가 이러한 값을 공유하도록 할 수 있음
- How? Broadcast!



## 14.2 언제 context를 써야 할까?(예제)

- React 컴포넌트 트리 안에서 전역(global) 데이터를 공유할 수 있도록 고안된 방법
  - 전역 데이터로는 현재 로그인한 유저, 테마, 선호하는 언어 등이 있음
- 여러 컴포넌트에서 계속해서 접근이 일어날 수 있는 데이터들이 있는 경우

## 14.3 Context API

- 1) React.createContext(defaultValue)
  - 컨텍스트를 생성하기 위한 함수
  - 컨텍스트 객체를 리턴함
  - defaultValue: Provider를 **찾지 못했을 때만** 쓰이는 값

```
const MyContext = React.createContext(defaultValue);
```

## 14.3 Context API

- 2) Context.Provider
  - 모든 컨텍스트 객체는 Provider라는 컴포넌트를 갖고 있음
  - Provider 컴포넌트로 하위 컴포넌트들을 감싸주면 모든 하위 컴포넌트들이 해당 컨텍스트의 데이터에 접근 가능
  - Provider에는 value라는 prop이 있으며, 하위에 있는 컴포넌트에게 전달됨
  - 여러 개의 Provider 중첩 가능

```
<MyContext.Provider value={/* 어떤 값 */}>
```



## 14.3 Context API

- 3) Context.Consumer
  - 컨텍스트의 데이터 변화를 구독하는 컴포넌트
  - 데이터를 소비한다는 뜻에서 consumer 컴포넌트라고도 부름
  - consumer 컴포넌트는 컨텍스트 값의 변화를 지켜보다가 값이 변경되면 재렌더링됨
  - 하나의 Provider 컴포넌트는 여러 개의 consumer 컴포넌트와 연결될 수 있음
  - 상위 레벨에 매칭되는 Provider가 없을 경우 기본값 사용

```
<MyContext.Consumer>  
  {value => /* context 값을 이용한 렌더링 */}  
</MyContext.Consumer>
```

## 14.3 Context API

- 4) Context.displayName
  - 크롬의 리액트 개발자 도구에서 표시되는 컨텍스트 객체의 이름

```
const MyContext = React.createContext(/* some value */);  
MyContext.displayName = 'MyDisplayName';  
  
<MyContext.Provider> // "MyDisplayName.Provider" in DevTools  
<MyContext.Consumer> // "MyDisplayName.Consumer" in DevTools
```

## 14.4 여러 개의 context 사용하기(예제)

- Provider 컴포넌트와 Consumer 컴포넌트를 여러 개 중첩해서 사용하면 됨

## 14.5 useContext

- 함수 컴포넌트에서 컨텍스트를 쉽게 사용할 수 있게 해주는 훅
- 생성된 **컨텍스트 객체**를 **인자**로 받아서 현재 컨텍스트의 값을 리턴
- 컨텍스트의 값이 변경되면 변경된 값과 함께 `useContext()`를 사용하는 컴포넌트가 재렌더링됨

```
const value = useContext(MyContext);
```

## 14.6 테마 변경 기능 만들기(실습)

- my-app/src/chapter14 실습

# 15. styled-components

CSS in JS 기술

# 일정 관리 앱 만들기

my-todo 실습

# 16. react-router-dom

page경로에 따른 라우팅



# 미니 블로그 만들기

my-blog 실습

# To-do 챌린지

리액트 개인 프로젝트  
To-do, Memo 등 일정 관리 앱 만들기

# 17. Axios

Promise 기반 HTTP 요청 라이브러리

# 뉴스 뷰어 만들기

my-news 실습

# 18. Redux

전역 상태 관리

# 쇼핑몰 만들기

my-shop 실습

- Redux Toolkit