# Component Style Guide

본 가이드는 React 프로젝트를 수행하는 UI 개발자 참고용으로 작성되었으며, 참여하는 프로젝트에서 일관된 컴포넌트 구조를 가지도록 컨벤션을 정하는데 활용할 수 있습니다.

## 목차

* [컴포넌트](#컴포넌트)
  + [컴포넌트 파일 구조](#컴포넌트-파일-구조)
  + [컴포넌트 문법 구조](#컴포넌트-문법-구조)
* [상태 값](#상태-값)
  + [컴포넌트를 신규로 만드는 경우](#컴포넌트를-신규로-만드는-경우)
  + [개발된 컴포넌트를 수정하는 경우](#개발된-컴포넌트를-수정하는-경우)
  + [컴포넌트 내부에 서브 컴포넌트가 있는 경우](#컴포넌트-내부에-서브-컴포넌트가-있는-경우)
  + [자식 컴포넌트에 상태 값을 전달하는 경우](#자식-컴포넌트에-상태-값을-전달하는-경우)
* [동적 랜더링](#동적-랜더링)
  + [CSS 클래스 조건 처리](#css-클래스-조건-처리)
  + [컴포넌트 노출의 조건 처리](#컴포넌트-노출의-조건-처리)
  + [노출할 컨텐츠를 컴포넌트 외부에서 관리하는 경우](#노출할-컨텐츠를-컴포넌트-외부에서-관리하는-경우)
* [목록의 처리](#목록의-처리)
  + [정적 목록](#정적-목록)
  + [동적 목록](#동적-목록)
  + [목록의 데이터 전달](#목록의-데이터-전달)

## 컴포넌트

컴포넌트의 구성에 대한 안내입니다.

### 컴포넌트 파일 구조

* 원칙은 해당 프로젝트의 가이드를 따릅니다. 특별한 가이드가 없는 경우 다음과 같이 컴포넌트 파일을 관리합니다.
* src  
  ├─── components // 컴포넌트 폴더  
  | ├─── ComponentName.js (컴포넌트)  
  | └─── ComponentName.scss (필요 시, 컴포넌트용 SCSS 파일)  
  | └─── ComponentName-dummy.js (필요 시, 컴포넌트용 테스트 props 모듈)  
  |  
  ├─── pages // 화면 단위의 React 페이지 컴포넌트 저장 폴더  
  └─── scss // 전역으로 사용되는 스타일들(reset, mixin, variables ...)
* 보통은 fe 개발자에게 pages 파일과 마크업 최상위 컨테이너를 만들어달라고 요청하는 것을 권장합니다.

### 컴포넌트 문법 구조

* 컴포넌트의 문법 구조는 함수형/클래스형 두 가지가 있습니다.
* 기본적으로 클래스형을 사용합니다.

#### 함수형

* + 변하는 데이터 없이 정적 마크업만 있을 때 사용됩니다.
  + 컴포넌트 내부에 선언하는 서브 컴포넌트에서 사용됩니다.
  + ES5 문법 예시
* // ex\_code\_1  
  import React from 'react';  
    
  const App = function () {  
  return (  
   <div>  
   <h2>함수형</h2>  
   <p>ES5 Type 문법</p>  
   </div>  
  );  
  }  
    
  export default App;
  + ES6 문법 예시 (화살표 함수/구조 간소화)
* // ex\_code\_2  
  import React from 'react';  
    
  const App = () => (  
   <div>  
   <h2>함수형</h2>  
   <p>ES6 Type 문법(화살표 함수)</p>  
   </div>  
  );  
    
  export default App;

#### 클래스형

* + 동적 랜더링이 필요한 컴포넌트를 만들 때 사용합니다.
* // ex\_code\_3  
  import React from 'react';  
    
  const App = () => (  
   <div>  
   <h2>함수형</h2>  
   <p>ES6 Type 문법(화살표 함수)</p>  
   </div>  
  );  
    
  export default App;
  + 필요한 데이터는 DEMO\_PROPS에 임시로 정의합니다.
* // ex\_code\_4  
  import React from 'react';  
    
  // 동적 처리에 필요한 데이터  
  const DEMO\_PROPS = {  
   title: '동적 처리에 필요한 데이터'  
  };  
    
  class App extends React.Component {  
   render() {  
   // 임시 데이터를 변수에 할당  
   const { title } = DEMO\_PROPS;  
    
   // 임시 데이터 적용  
   return (  
   <h2>{title}</h2>  
   );  
   }  
  }  
    
  export default App;
* 컴포넌트 내부에 선언하는 서브 컴포넌트는 동적인 랜더링 여부와 상관 없이 함수형으로 사용합니다.
* //ex\_code\_5  
  import React from 'react';  
    
  const DEMO\_PROPS = {  
   title: '클래스형 예제'  
  };  
    
  // 서브 컴포넌트의 DEMO\_PROPS  
  const DEMO\_PROPS\_SUB\_COMPONENT2 = {  
   title: '서브 컴포넌트 2'  
  };  
    
  // DEMO\_PROPS 를 사용하지 않는 서브 컴포넌트  
  const SubComponent1 = () => <p>서브 컴포넌트 1</p>;  
    
  // DEMO\_PROPS 를 사용하는 서브 컴포넌트  
  const SubComponent2 = () => {  
   const { title } = DEMO\_PROPS\_SUB\_COMPONENT2;  
    
   return <p>{ title }</p>  
  };  
    
  class App extends React.Component {  
   render() {  
   const { title } = DEMO\_PROPS;  
    
   return (  
   <div>  
   <SubComponent1 />  
   <SubComponent2 />  
   </div>  
   );  
   }  
  }  
    
  export default App;

## 상태 값

랜더링 처리에 필요한 상태 값을 관리하는 방법을 안내합니다.

상태 값이란 쉽게 말해서 렌더링에 영향을 주는 데이터라고 할 수 있습니다. 일반적으로 컴포넌트 외부에서 접근할 수 없는 상태 값은 state로 관리하고, 외부에서 제어하는 상태 값은 props로 관리합니다. 마크업 개발 단계에서는 비즈니스 로직을 예상할 수 없기 때문에 이 둘을 구분하지 않습니다.

### 컴포넌트를 신규로 만드는 경우

* 컴포넌트에서 상태 값이 필요한 경우 import 구문 바로 아래에 DEMO\_PROPS를 추가하여 데이터를 관리합니다.
* //ex\_code\_6  
  import React from 'react';  
    
  // #1 마크업 개발용 데이터 정의  
  const DEMO\_PROPS = {  
   id: '신규 컴퍼넌트 상태값 적용하기',  
   title: 'import 구문 바로 아래에 DEMO\_PROPS를 추가하여 데이터를 관리합니다.'  
  };  
    
  class App extends React.Component {  
   render() {  
   // #2 props용 변수에 값 할당  
   const { id, title } = DEMO\_PROPS;  
    
   // #3 데이터 적용  
   return (  
   <div>  
   <p>{id}</p>  
   <p>{title}</p>  
   </div>  
   );  
   }  
  }  
    
  export default App;

### 개발된 컴포넌트를 수정하는 경우

* 개발된 컴포넌트에서 state나 props에 필요한 값이 없는 경우, 해당 값만 DEMO\_PROPS로 추가하여 사용합니다.
* import React from 'react';  
    
  class App extends React.Component<Props> {  
   render() {  
   const { id, title } = this.props;  
    
   return (  
   <div>  
   <p>{id}</p>  
   <p>{title}</p>  
   </div>  
   );  
   }  
  }  
    
  export default App;
* import React from 'react';  
    
  // #1 마크업 개발용 데이터 추가  
  const DEMO\_PROPS = {  
   preview: '미리 보기 내용'  
  };  
    
  class App extends React.Component<Props> {  
   render() {  
   // #2 기존의 props의 위치와 상관 없이 첫 번째 라인에 별도로 추가  
   const { preview } = DEMO\_PROPS;  
    
   const { id, title } = this.props;  
    
   return (  
   <div>  
   <p>{id}</p>  
   <p>{title}</p>  
    
   {/\* #3 추가 데이터 반영 \*/}  
   <p>{preview}</p>  
   </div>  
   );  
   }  
  }  
    
  export default App;

### 컴포넌트 내부에 서브 컴포넌트가 있는 경우

* DEMO\_PROPS외에 DEMO\_PROPS\_SUB\_COMPONENT 형태로 별도 추가하여 관리합니다.
* //ex\_code\_7  
  import React from 'react';  
    
  const DEMO\_PROPS = {  
   id: '컴퍼넌트 IN 컴퍼넌트',  
   title: '마크업 데모 데이터'  
  };  
    
  // #1 서브 컴포넌트 개발용 데이터 정의  
  const DEMO\_PROPS\_BADGE = {  
   type: 'admin'  
  };  
    
  const Badge = () => {  
   // #2 서브 컴포넌트의 props를 변수에 할당  
   const { type } = DEMO\_PROPS\_BADGE;  
    
   return type === 'admin' && <p>Badge admin</p>  
  };  
    
  class App extends React.Component {  
   render() {  
   const { id, title } = DEMO\_PROPS;  
    
   return (  
   <div>  
   <h2>{id}</h2>  
   <p>{title}</p>  
   {/\* #3 서브 컴포넌트 적용 \*/}  
   <Badge />  
   </div>  
   );  
   }  
  }  
    
  export default App;

### 자식 컴포넌트에 상태 값을 전달하는 경우

* 자식 컴포넌트에서 부모 컴포넌트의 상태 값 사용이 필요한 경우 props로 전달합니다.

React UI 개발 가이드에서는 랜더링에 필요한 임시 데이터는 해당 컴포넌트 내부에서 관리하는 것을 권장합니다.

**App**

* import React from 'react';  
  import AppChild from './AppChild';  
    
  // #1 자식 컴포넌트와 공유할 category 값을 부모 컴포넌트에서 정의  
  const DEMO\_PROPS = {  
   category: 'food',  
   title: '자식 컴포넌트 상태 값 전달'  
  };  
    
  class App extends React.Component {  
   render() {  
   const { title, category } = DEMO\_PROPS;  
    
   return (  
   <div>  
   <h2>{title}</h2>  
    
   {/\* #2 자식 컴포넌트의 props로 category 상태 값 전달 \*/}  
   <AppChild category={category} />  
   </div>  
   );  
   }  
  }  
    
  export default App;

**AppChild**

* import React from 'react';  
    
  const DEMO\_PROPS = {  
   contents: '자식 컴포넌트에서 부모 컴포넌트의 상태 값 사용'  
  };  
    
  class AppChild extends React.Component {  
   render() {  
   const { contents } = DEMO\_PROPS;  
    
   // #3 부모에서 전달하는 category 값은 DEMO\_PROPS 가 아닌 this.props 를 통해서 사용  
   const { category } = this.props;  
    
   // 카테고리명-wrap 으로 클래스 가공  
   const wrapClass = `${category}-wrap`;  
    
   return (  
   // 클래스 적용  
   <div className={wrapClass}>  
   <p>{contents}</p>  
   </div>  
   );  
   }  
  }  
    
  export default AppChild;
* 컴포넌트 내에 있는 서브 컴포넌트에 props를 전달하는 경우는 파라미터로 받아서 처리합니다.
* // ex\_code\_9  
  import React from 'react';  
    
  // #1 자식 컴포넌트와 공유할 category 값을 부모 컴포넌트에서 정의  
  const DEMO\_PROPS = {  
   category: 'food',  
   title: '컨퍼넌트 IN 컴퍼넌트 props 전달',  
   content: '파라미터로 받아서 처리'  
  };  
    
  // #2 서브 컴포넌트(함수형)의 파라미터에 props를 추가하고 변수에 할당  
  const SubComponent = (props) => {  
   const { category, content } = props;  
    
   // #3 props 적용  
   return (  
   <div className={category}>  
   <p>{content}</p>  
   </div>  
   );  
  };  
    
  class App extends React.Component {  
   render() {  
   const { category, title, content } = DEMO\_PROPS;  
    
   return (  
   <div>  
   <h2>{title}</h2>  
    
   {/\* #4 서브 컴포넌트에 props 전달 \*/}  
   <SubComponent  
   category={category}  
   content={content}  
   />  
   </div>  
   );  
   }  
  }  
    
  export default App;
* props를 활용하게 되면 해당 컴포넌트를 사용할 때 항상 부모 컴포넌트에서 해당 값을 전달해야 합니다. 이 경우 defaultProps를 이용하면 값을 전달하지 않았을 때 사용할 값을 지정할 수 있습니다.

defaultProps의 설정 방법은 프로젝트의 react 세팅에 따라 여러가지 방법이 존재합니다.

**일반적인 환경**

* //ex\_code\_10  
  import React from 'react';  
    
  class AppChild extends React.Component {  
   render() {  
   // this.props는 부모 컴포넌트가 전달하는 값  
   const { title } = this.props;  
    
   return (  
   // 클래스 적용  
   <div>  
   <p>{title}</p>  
   </div>  
   );  
   }  
  }  
    
  // title props에 대한 기본 값 설정  
  AppChild.defaultProps = {  
   title: '제목이 없습니다.'  
  };  
    
  export default AppChild;

**Proposal Class Properties 플러그인 사용 환경**

[babel-plugin-proposal-class-properties](https://babeljs.io/docs/en/babel-plugin-proposal-class-properties)

* //ex\_code\_11  
  import React from 'react';  
    
  class AppChild extends React.Component {  
    
   // title props에 대한 기본 값 설정  
   static defaultProps = {  
   title: '제목이 없습니다.'  
   };  
    
   render() {  
   // this.props는 부모 컴포넌트가 전달하는 값  
   const { title } = this.props;  
    
   return (  
   // 클래스 적용  
   <div>  
   <p>{title}</p>  
   </div>  
   );  
   }  
  }  
    
  export default AppChild;

## 동적 랜더링

조건에 따라 템플릿을 처리하는 방법을 안내합니다.

### CSS 클래스 조건 처리

* 조건에 따른 클래스 처리는 항상 [classnames](https://www.npmjs.com/package/classnames) 라이브러리를 사용하고 별도의 변수로 처리합니다.

**기본 처리**

* //ex\_code\_12  
  import React from 'react';  
  import classnames from 'classnames';  
    
  const DEMO\_PROPS = {  
   isLogin: true  
  };  
    
  class App extends React.Component {  
   render() {  
   const { isLogin } = DEMO\_PROPS;  
    
   const expandClass = classnames({  
   // 로그인 상태가 아닌 경우 expand 클래스 노출  
   'expand': !isLogin  
   });  
    
   return (  
   // 로그인을 안한 상태면 expand 클래스 추가  
   <div className={expandClass}>  
   <h2>CSS 클래스 조건 처리</h2>  
   </div>  
   );  
   }  
  }  
    
  export default App;

**복잡한 조건 처리**

* //ex\_code\_13  
  import React from 'react';  
  import classnames from 'classnames';  
    
  const DEMO\_PROPS = {  
   isLogin: false,  
   userType: 'admin'  
  };  
    
  class App extends React.Component {  
   render() {  
   const { isLogin, userType } = DEMO\_PROPS;  
    
   const expandClass = classnames({  
   'expand': !isLogin, // 로그인이 아닌 경우라면 expand 클래스 노출  
   'info-wrap': userType !== 'admin' // admin이 아니면 info-wrap 클래스 노출  
   });  
    
   const loginClass = classnames({  
   'login': isLogin, // 로그인이면 login 클래스 노출  
   'user-admin': userType === 'admin', // admin이면 user-admin 클래스 노출  
   'user-staff': userType !== 'admin' // admin이 아니면 user-staff 클래스 노출  
   });  
    
   return (  
   // 조건 처리된 클래스 적용  
   <div className={expandClass}>  
   <h2 className={loginClass}>CSS 복잡한 조건 처리</h2>  
   </div>  
   );  
   }  
  }  
    
  export default App;
* 조건과 상관 없는 고정 클래스가 있는 경우는 classsnames의 첫 번째 파라미터로 전달합니다.
* //ex\_code\_14  
  import React from 'react';  
  import classnames from 'classnames';  
    
  const DEMO\_PROPS = {  
   isLogin: false  
  };  
    
  class App extends React.Component {  
   render() {  
   const { isLogin } = DEMO\_PROPS;  
    
   // box, item 클래스는 항상 노출하고 로그인 한 상태에 따라 expand 클래스 노출  
   const expandClass = classnames('box item', {  
   'expand': !isLogin  
   });  
    
   return (  
   // 로그인 상태면 'box item'만 출력하고 로그인이 아니면 'box item expand'를 출력  
   <div className={expandClass}>  
   <h2>고정 클래스가 있는 경우</h2>  
   </div>  
   );  
   }  
  }  
    
  export default App;

### 컴포넌트 노출의 조건 처리

* 조건 따른 컴포넌트를 노출: && 연산자

**컴포넌트 조건부 랜더링**

* //ex\_code\_15  
  import React from 'react';  
    
  const DEMO\_PROPS = {  
   isLogin: true  
  };  
    
  class App extends React.Component {  
   render() {  
   const { isLogin } = DEMO\_PROPS;  
    
   // 로그인 상태에서만 컴포넌트 노출  
   return isLogin && (  
   <div>  
   <h2>컴포넌트 조건부 랜더링</h2>  
   </div>  
   );  
   }  
  }  
    
  export default App;

**특정 영역 조건부 랜더링**

* //ex\_code\_16  
  import React from 'react';  
    
  const DEMO\_PROPS = {  
   isLogin: true  
  };  
    
  class App extends React.Component {  
   render() {  
   const { isLogin } = DEMO\_PROPS;  
    
   return (  
   <div>  
   <h2>특정 영역 조건부 랜더링</h2>  
   {/\* 로그인 상태에서만 표시 \*/}  
   {isLogin && (  
   <p>  
   로그인 상태에서만 노출  
   </p>  
   )}  
   </div>  
   );  
   }  
  }  
    
  export default App;
* 조건 따른 컴포넌트를 노출: 삼항연산자

**컴포넌트 조건부 랜더링**

* //ex\_code\_17  
  import React from 'react';  
    
  const DEMO\_PROPS = {  
   isLogin: true,  
   userType: 'admin'  
  };  
    
  class App extends React.Component {  
   render() {  
   const { isLogin, userType } = DEMO\_PROPS;  
    
   // admin 유저가 로그인한 상태  
   const visible = isLogin && userType === 'admin';  
    
   // admin 유저가 로그인 했을 때만 컴포넌트 출력  
   return visible ? (  
   <div>  
   <h2>컴퍼넌트 조건부 랜더링</h2>  
   <p>admin 유저가 로그인 했을 때</p>  
   </div>  
   ) : (  
   <div>  
   <h2>컴퍼넌트 조건부 랜더링</h2>  
   <p>admin 유저가 로그인 안 했을 때</p>  
   </div>  
   );  
   }  
  }  
    
  export default App;

**특정 영역 조건부 랜더링**

* //ex\_code\_18  
  import React from 'react';  
    
  const DEMO\_PROPS = {  
   isLogin: true  
  };  
    
  class App extends React.Component {  
   render() {  
   const { isLogin } = DEMO\_PROPS;  
    
   return (  
   <div>  
   <h2> 특정 영역 조건부 랜더링 </h2>  
   {/\* 로그인 상태에서만 표시 \*/}  
   <p>  
   {/\* 로그인 했을 때만 표시 \*/}  
   {isLogin ?  
   <span>로그인 했을 때</span> : <span>로그인 안 했을 때</span>  
   }  
   </p>  
   </div>  
   );  
   }  
  }  
    
  export default App;
* 조건 따른 컴포넌트를 노출: 노출하는 영역이 복잡하다면 별도의 변수나 컴포넌트로 관리

**변수에 담아서 처리하는 방법**

* //ex\_code\_19  
  import React from 'react';  
    
  const DEMO\_PROPS = {  
   isLogin: true  
  };  
    
  class App extends React.Component {  
   render() {  
   const { isLogin } = DEMO\_PROPS;  
    
   // 로그인 일 때 노출  
   const userImage = isLogin && (  
   <p>  
   로그인 일 때 노출  
   </p>  
   );  
    
   return (  
   <div>  
   <h2> 변수에 담아서 처리하는 방법 </h2>  
   {/\* 로그인 상태에서만 표시 \*/}  
   {userImage}  
   </div>  
   );  
   }  
  }  
    
  export default App;

**별도의 컴포넌트로 처리하는 방법**

* // ex\_code\_20  
  import React from 'react';  
    
  const DEMO\_PROPS = {  
   isLogin: true  
  };  
    
  // 로그인 상태에서만 노출 할 컴포넌트  
  const SubComponent = (props) => {  
   const { isLogin } = props;  
    
   return isLogin && (  
   <p>로그인 일 때 노출</p>   
   );  
  };  
    
  class App extends React.Component {  
   render() {  
   const { isLogin } = DEMO\_PROPS;  
    
   return (  
   <div>  
   <p> 별도의 컴포넌트로 처리하는 방법 </p>  
   {/\* 로그인 상태에서만 표시 \*/}  
   <SubComponent isLogin={isLogin} />  
   </div>  
   );  
   }  
  }  
    
  export default App;
* 노출 조건이 복잡한 경우는 별도의 변수로 관리합니다.

**컴포넌트 조건부 랜더링**

* //ex\_code\_21  
  import React from 'react';  
    
  const DEMO\_PROPS = {  
   isLogin: true,  
   userType: 'admin'  
  };  
    
  class App extends React.Component {  
   render() {  
   const { isLogin, userType } = DEMO\_PROPS;  
    
   // admin 유저가 로그인한 상태  
   const visible = isLogin && userType === 'admin';  
    
   // admin 유저가 로그인 했을 때만 컴포넌트 출력  
   return visible && (  
   <div>  
   <p>노출 조건이 복잡한 경우는 별도의 변수로 관리합니다.</p>  
   <h2>컴포넌트 조건부 랜더링</h2>  
   <p>admin 유저가 로그인 했을 때만 컴포넌트 출력</p>  
   </div>  
   );  
   }  
  }  
    
  export default App;

**특정 영역 조건부 랜더링**

* //ex\_code\_22  
  import React from 'react';  
    
  const DEMO\_PROPS = {  
   isLogin: true,  
   userType: 'admin'  
  };  
    
  class App extends React.Component {  
   render() {  
   const { isLogin, userType } = DEMO\_PROPS;  
    
   // admin 유저가 로그인한 상태  
   const visible = isLogin && userType === 'admin';  
    
   return (  
   <div>  
   <p>노출 조건이 복잡한 경우는 별도의 변수로 관리합니다.</p>  
   <h2>컴포넌트 조건부 랜더링</h2>  
   {/\* admin 유저가 로그인 했을 때만 표시 \*/}  
   {visible && (  
   <p>admin 유저가 로그인 했을 때만 컴포넌트 출력</p>  
   )}  
   </div>  
   );  
   }  
  }  
  export default App;
* 출력하는 조건이 여러가지면 switch 구문으로 관리합니다.

**컴포넌트 조건부 랜더링**

* //ex\_code\_23  
  import React from 'react';  
    
  const DEMO\_PROPS = {  
   userType: 'admin'  
  };  
    
  class App extends React.Component {  
   render() {  
   const { userType } = DEMO\_PROPS;  
    
   // 조건 별로 컴포넌트 출력  
   switch(userType) {  
   case 'admin' :  
   return (  
   <h2>관리자인 경우</h2>  
   );  
   case 'user' :  
   return (  
   <h2>사용자인 경우</h2>  
   );  
   default :  
   /\*  
   // 기본 값이 없는 경우  
   return null;  
   \*/  
   return (  
   <h2>값이 없는 경우</h2>  
   );  
   }  
   }  
  }  
    
  export default App;

**특정 영역 조건부 랜더링**

* //ex\_code\_24  
  import React from 'react';  
    
  const DEMO\_PROPS = {  
   userType: 'admin'  
  };  
    
  class App extends React.Component {  
   render() {  
   const { userType } = DEMO\_PROPS;  
    
   let userInfo = null;  
    
   // 조건 별로 유저 정보 표시  
   switch(userType) {  
   case 'admin' :  
   userInfo = (  
   <p>관리자인 경우</p>  
   );  
   break;  
   case 'user' :  
   userInfo = (  
   <p>사용자인 경우</p>  
   );  
   break;  
   default :  
   // 기본 값이 없으면 break만 처리  
   userInfo = (  
   <p>조건이 없는 경우</p>  
   );  
   break;  
   }  
    
   return (  
   <div>  
   <h2>특정 영역 조건부 랜더링</h2>  
   {/\* 유저 종류에 따라 표시 \*/}  
   {userInfo}  
   </div>  
   );  
   }  
  }  
    
  export default App;

### 노출할 컨텐츠를 컴포넌트 외부에서 관리하는 경우

* props.children을 이용하여 컴포넌트 외부에서 주입하는 내용을 출력할 수 있습니다.

**ParentComponent**

* //ex\_code\_25  
  import React from 'react';  
  import AppChild from './AppChild';  
    
  class App extends React.Component {  
   render() {  
   return (  
   // #1 Box 컴포넌트에서 출력할 내용을 부모 컴포넌트가 전달  
   <Box>  
   <p>Box 컴포넌트에서 출력할 내용</p>  
   <p>Box 컴포넌트에서 출력할 내용</p>  
   <p>Box 컴포넌트에서 출력할 내용</p>  
   </Box>  
   );  
   }  
  }  
    
  export default App;

**AppChild**

* //ex\_code\_25-2  
  import React from 'react';  
    
  class AppChild extends React.Component {  
   render() {  
   return (  
   <div className="box">  
   <div className="content">  
   {/\* #2 this.props.children을 이용하여 외부에서 입력한 내용을 출력 \*/}  
   {this.props.children}  
   </div>  
    
   <button>내용 펼치기</button>  
   </div>  
   );  
   }  
  }  
    
  export default AppChild;

## 목록의 처리

목록을 처리하는 방법을 안내합니다.

### 정적 목록

* 데이터를 이용하여 출력하지 않아도 큰 문제가 없는 경우는 정적으로 처리합니다. 데이터로 출력되는 목록이라도 간단한 경우라면 정적으로 처리합니다.
* //ex\_code\_26  
  import React from 'react';  
    
  const App = () => (  
   <ul>  
   <li>목록</li>  
   <li>목록</li>  
   <li>목록</li>  
   </ul>  
  );  
    
  export default App;

### 동적 목록

* 기본적으로 동적으로 처리할 배열의 요소는 id 속성만 가진 객체를 사용합니다. 이외의 값은 정의하지 않습니다. > 목록의 key를 설정하지 않으면 빌드 시점에 경고문이 출력됩니다. > [(key 값에 배열의 index를 사용하지 않는 이유)](https://reactjs-kr.firebaseapp.com/docs/reconciliation.html#recursing-on-children)
* const DEMO\_PROPS = {  
   itemList: [  
   {id: 1},  
   {id: 2},  
   {id: 3}  
   ]  
  };
* 배열 데이터에 따라 반복 처리는 Array.prototype.map 메서드를 사용합니다.
* import React from 'react';  
    
  const DEMO\_PROPS = {  
   itemList: [  
   ]  
  };  
    
  class App extends React.Component {  
   render() {  
   const { itemList } = DEMO\_PROPS;  
    
   return (  
   <ul>  
   {/\*#1 map을 이용하여 필요한 위치에 목록 출력\*/}  
   {itemList.map((item) => (  
   // #2 key 추가  
   <li key={item.id}>목록</li>  
   ))}  
   </ul>  
   );  
   }  
  }  
    
  export default App;
* 배열 데이터가 없는 조건은 배열의 length 값으로 판단합니다.

**목록 아이템 노출을 조건 처리 하는 경우**

* //ex\_code\_27  
  import React from 'react';  
    
  const DEMO\_PROPS = {  
   itemList: []  
  };  
    
  class App extends React.Component {  
   render() {  
   const { itemList } = DEMO\_PROPS;  
    
   // 배열 데이터가 있는 조건  
   const visibleItemList = itemList && itemList.length > 0;  
    
   // 아이템을 조건에 따라 노출  
   return (  
   <ul>  
   {visibleItemList ? (  
   // 배열이 노출되는 조건  
   itemList.map((item) => (  
   // #2 key 추가  
   <li key={item.id}>목록</li>  
   ))  
   ) : (  
   // 배열이 노출되지 않는 조건  
   <li>목록이 없습니다.</li>  
   )}  
   </ul>  
   );  
   }  
  }  
    
  export default App;

**목록 자체 노출을 조건 처리 하는 경우**

* //ex\_code\_28  
  import React from 'react';  
    
  const DEMO\_PROPS = {  
   itemList: []  
  };  
    
  class App extends React.Component {  
   render() {  
   const { itemList } = DEMO\_PROPS;  
    
   // 배열 데이터가 있는 조건  
   const visibleItemList = itemList && itemList.length > 0;  
    
   // 배열을 조건에 따라 노출  
   return visibleItemList ?  
   (  
   // 배열이 노출되는 조건  
   <ul>  
   {itemList.map((item) => (  
   // #2 key 추가  
   <li key={item.id}>목록</li>  
   ))}  
   </ul>  
   ) : (  
   // 배열이 노출되지 않는 조건  
   <div>목록이 없습니다.</div>  
   );  
   }  
  }  
    
  export default App;
* 컴포넌트를 반복해서 출력하는 경우 반복되는 요소를 별도의 컴포넌트로 관리할 수 있습니다. > 반복되는 요소의 상태 값은 부모 컴포넌트의 배열이 아닌 해당 컴포넌트 내에서 관리하는 것을 권장합니다.

**App**

* //ex\_code\_29-1  
  import React from 'react';  
  import AppChild from './AppChild';  
    
  const DEMO\_PROPS = {  
   itemList: [  
   {id: 1},  
   {id: 2},  
   {id: 3}  
   ]  
  };  
    
  class App extends React.Component {  
   render() {  
   const { itemList } = DEMO\_PROPS;  
    
   return (  
   <ul>  
   {/\*#1 map을 이용하여 필요한 위치에 목록 출력\*/}  
   {itemList.map((item) => (  
   // #2 반복되는 최상위 요소에 key 추가  
   <li key={item.id}>  
   {/\*#3 반복할 컴포넌트 추가\*/}  
   <AppChild />  
   </li>  
   ))}  
   </ul>  
   );  
   }  
  }  
    
  export default App;

**AppChild**

* //ex\_code\_29-2  
  import React from 'react';  
    
  const DEMO\_PROPS = {  
   id: 1234,  
   title: '목록의 제목'  
  };  
    
  class AppChild extends React.Component {  
   render() {  
   const { id, title } = DEMO\_PROPS;  
    
   return (  
   <React.Fragment>  
   <p>{id}</p>  
   <p>{title}</p>  
   </React.Fragment>  
   );  
   };  
  }  
    
  export default AppChild;

### 목록의 데이터 전달

컴포넌트에서 필요한 데이터는 컴포넌트 내부에서 관리하는 것을 권장하지만, 필요한 경우 반복되는 엘리먼트에서 처리할 상태 값을 배열 데이터에서 관리할 수 있습니다.

* 반복 되는 요소가 컴포넌트가 아닌 경우는 직접 데이터를 사용합니다.
* //ex\_code\_30  
  import React from 'react';  
    
  // #1 목록 데이터 정의  
  const DEMO\_PROPS = {  
   itemList: [  
   {  
   id: 1,  
   title: '제목 1'  
   },  
   {  
   id: 2,  
   title: '제목 2'  
   },  
   {  
   id: 3,  
   title: '제목 3'  
   }  
   ]  
  };  
    
  class ListComponent extends React.Component {  
   render() {  
   const { itemList } = DEMO\_PROPS;  
    
   return (  
   <ul>  
   {itemList.map((item) => (  
   // #2 key를 추가하고 item 값을 직접 접근해서 사용  
   <li key={item.id}>  
   <p>{item.title}</p>  
   </li>  
   ))}  
   </ul>  
   );  
   }  
  }  
    
  export default ListComponent;
* 반복 요소로 별도의 컴포넌트를 사용하는 경우 props로 데이터를 전달합니다.

**App**

* //ex\_code\_31-1  
  import React from 'react';  
  import AppChild from './AppChild';  
    
  // #1 목록 데이터 정의  
  const DEMO\_PROPS = {  
   itemList: [  
   {  
   id: 1,  
   title: '제목 1'  
   },  
   {  
   id: 2,  
   title: '제목 2'  
   },  
   {  
   id: 3,  
   title: '제목 3'  
   }  
   ]  
  };  
    
  class App extends React.Component {  
   render() {  
   const { itemList } = DEMO\_PROPS;  
    
   return (  
   <ul>  
   {itemList.map((item) => (  
   // #2 key를 추가하고 item 속성들을 스프레드 연산자를 이용해 props로 전달  
   <AppChild key={item.id} {...item} />  
   ))}  
   </ul>  
   );  
   }  
  }  
    
  export default App;

**AppChild**

* //ex\_code\_31-2  
  import React from 'react';  
    
  class AppChild extends React.Component {  
   render() {  
   // #3 DEMO\_PROPS 대신 this.props로 부모 컴포넌트가 전달하는 값을 직접 사용  
   const { id, title } = this.props;  
    
   return (  
   <li>  
   <p>{id}</p>  
   <p>{title}</p>  
   </li>  
   );  
   };  
  }  
    
  export default AppChild;
* 반복 요소가 서브 컴포넌트인 경우 파라미터로 props를 전달 받습니다.
* //ex\_code\_32  
  import React from 'react';  
    
  // #1 목록 데이터 정의  
  const DEMO\_PROPS = {  
   itemList: [  
   {  
   id: 1,  
   title: '제목 1'  
   },  
   {  
   id: 2,  
   title: '제목 2'  
   },  
   {  
   id: 3,  
   title: '제목 3'  
   }  
   ]  
  };  
    
  // #2 서브 컴포넌트에 props 파라미터 추가  
  const AppChild = (props) => {  
   const { title } = props;  
    
   // #3 props 적용  
   return <li>{title}</li>;  
  };  
    
  class App extends React.Component {  
   render() {  
   const { itemList } = DEMO\_PROPS;  
    
   return (  
   <ul>  
   {itemList.map((item) => (  
   // #4 key를 추가하고 item 속성들을 스프레드 연산자를 이용해 props로 전달  
   <AppChild key={item.id} {...item} />  
   ))}  
   </ul>  
   );  
   }  
  }  
    
  export default App;