

React를 이용한 프론트 엔드 개발

React 소개

“ A JavaScript library for building user interfaces ”

- ▶ Reference

- ▶ <https://ko.reactjs.org/>
- ▶ <https://ko.reactjs.org/docs/getting-started.html>

- ▶ React : 웹 페이지 화면을 개발하기 위한 라이브러리이자 프레임워크

- ▶ React를 이해하기 위한 세 가지 기본 개념

- ▶ Component
- ▶ JSX
- ▶ Props and State (데이터)

React의 장점

▶ 컴포넌트 기반

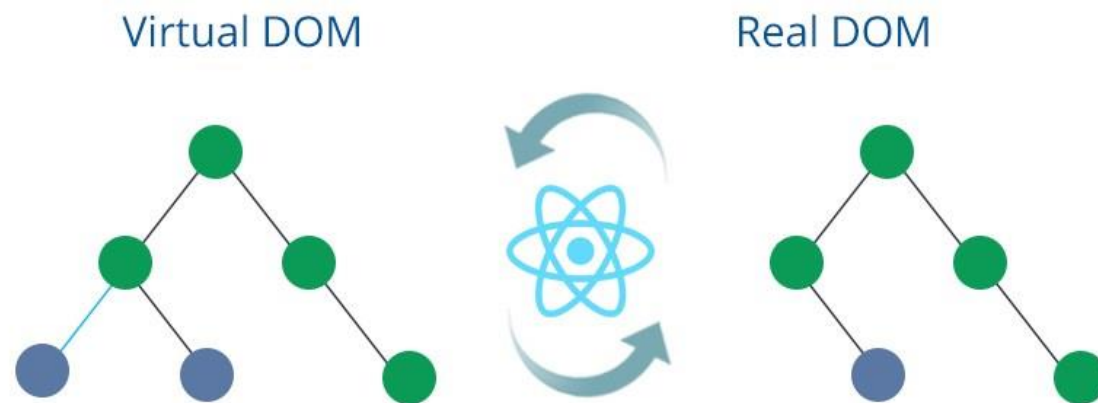
- ▶ 작고 독립적인 코드블록(컴포넌트)을 이용하여 빠르고 효율적으로 화면을 구성
- ▶ 잘 개발된 단위 컴포넌트는 재사용될 수 있다

▶ Virtual DOM

- ▶ DOM을 직접 제어하게 되면 화면이 복잡해 질수록 화면을 그리는 시간도 길어진다
- ▶ React는 화면의 노드들을 미리 그려 두고 변경된 부분만 실제 DOM에 적용하는 방식을 채택(Virtual DOM), 화면의 렌더링 속도를 극적으로 높임

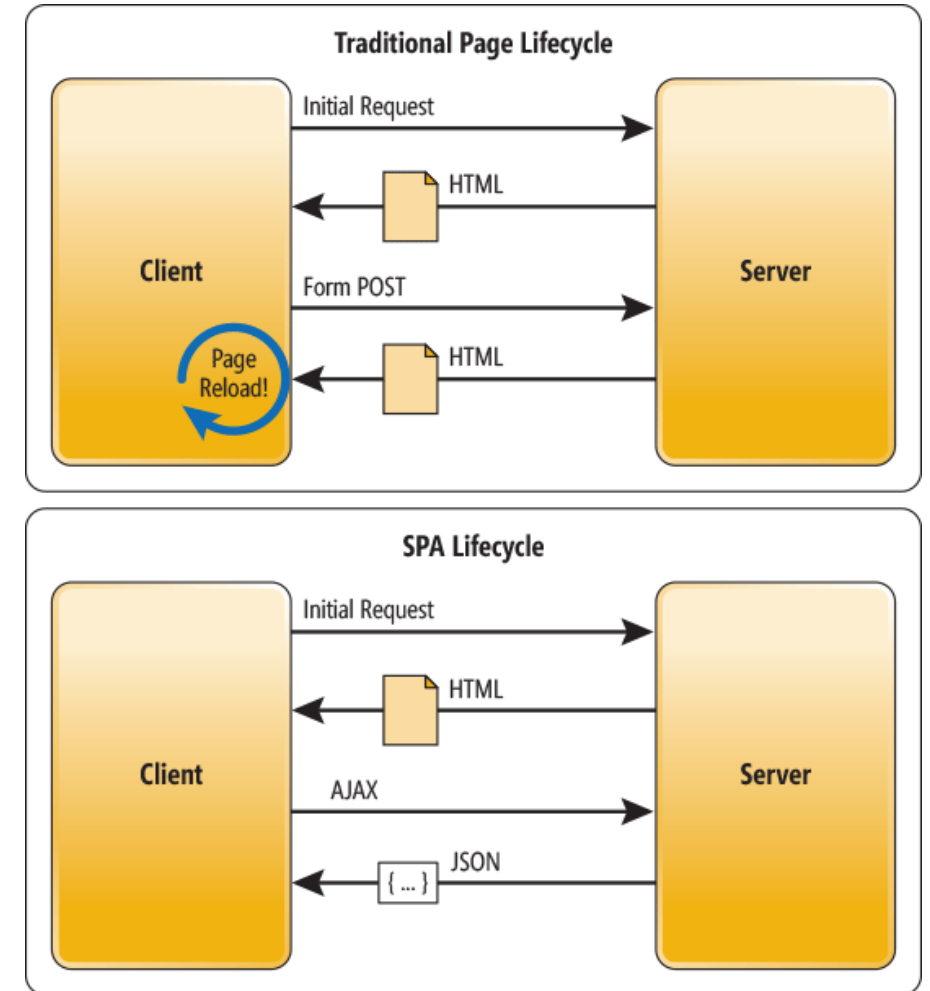
▶ 선언형(Declarative)

- ▶ 응용프로그램의 각 상태에 대한 뷰를 설계
- ▶ 데이터 변경시 적절한 컴포넌트만 갱신하고 렌더링



Single Page Application

- ▶ SPA : 기존 웹 개발 방식과는 다른 방식
 - ▶ Traditional Way : 웹 서버가 전체 페이지를 전송하고 브라우저는 단순 렌더링만 하던 방식
 - ▶ SPA Way : 브라우저가 전체 페이지를 매번 렌더링하지 않고 서버는 필요한 정보만 브라우저에 전송, 브라우저는 해당 부분만 다시 렌더링하는 방식
- ▶ SPA의 장점
 - ▶ App과 비슷한 사용자 경험을 제공할 수 있음
 - ▶ Flickering 없음
 - ▶ 유지보수의 편의성
 - ▶ 본격적인 Front-End와 Back-End의 역할 분담과 협업
 - ▶ 점진적 개발이 가능
 - ▶ 컴포넌트 단위의 개발 및 조합



Traditional Page vs SPA

How React Works?

기본 개념의 이해

How React Works?

- ▶ 리액트 라이브러리를 사용하는 가장 기본적인 방법:

- ▶ 2개의 자바스크립트를 읽어 들이는 것

index.html

```
<script src="https://unpkg.com/react@17/umd/react.development.js" crossorigin></script>  
<script src="https://unpkg.com/react-dom@17/umd/react-dom.development.js" crossorigin></script>
```

- ▶ References: <https://ko.reactjs.org/docs/cdn-links.html>

How React Works?

- ▶ React는 점진적으로 도입할 수 있도록 설계
 - ▶ 기존 HTML 사이트에 리엑트를 도입해 보면서 React를 이해해 봅시다.

js/index.js

```
function HelloButton() {  
  return React.createElement("button", // element  
    {  
      // ... options  
    },  
    "React Button"); // children  
}  
  
const container = document.getElementById("root");  
ReactDOM.render(React.createElement(HelloButton),  
  container);
```

index.html

```
...  
<div id="root"></div>  
...  
  
<!-- import React  
Library Script -->  
  
<script src="./js/index.js">  
</script>
```

- ▶ 만약... 복잡한 중첩 구조의 페이지를 위 방법으로 만들어야 한다면... ?

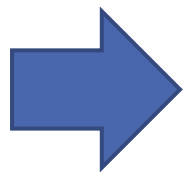
How React Works?

: JSX

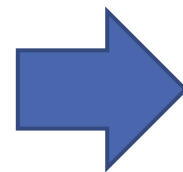
- ▶ 만약... 복잡한 중첩 구조의 페이지를 위 방법으로 만들어야 한다면... ?
 - ▶ `React.createElement` 함수를 중첩시켜 작성해야 한다
 - ▶ 가독성이 매우 떨어지며 컴포넌트의 구조를 파악하기 매우 어렵게 된다
- ▶ HTML 태그와 비슷한 방식으로 JavaScript를 작성할 수 있다면?
 - ▶ JSX(JavaScript eXtension) : Babel에 의해 JavaScript로 Transpile 된다
 - ▶ <https://babeljs.io/> 에서 복잡한 중첩 구조의 컴포넌트를 구성하고 어떤 방식으로 변환되는지 확인해 보시다.

```
<div>  
  <h1>My React Button</h1>  
  <button>My Button</button>  
</div>
```

JSX



BABEL



?

Vanilla JS

<https://ko.reactjs.org/docs/introducing-jsx.html>

How React Works?

: JSX

- ▶ JSX는 표준 ECMAScript가 아닙니다.
 - ▶ 반드시 Transpile 과정을 거쳐서 표준 JS로의 변환을 거쳐야 합니다.

- ▶ 기존 프로젝트에서 JSX 문법을 이용하고자 한다면
 - ▶ 페이지에 다음 스크립트 태그를 삽입

```
<script src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>
```

- ▶ 위 스크립트가 삽입된 이후에는 어떤 <script> 태그에서든 type="text/babel" 속성을 추가하면 JSX를 활용할 수 있다
 - ▶ 주의: 이 방식은 학습 목적 혹은 간단한 데모 사이트를 만들기에는 괜찮지만 사이트를 느리게 만들고 프로덕션용으로는 적합하지 않음

How React Works?

: 사용자 컴포넌트 만들기

- ▶ 사용자 컴포넌트를 만드는 두 가지 방법
 - ▶ class 컴포넌트
 - ▶ functional 컴포넌트 (Stateless Component)
- ▶ 클래스 컴포넌트 생성
 - ▶ React.Component를 상속받아 클래스를 생성
 - ▶ render() 메서드에서 JSX를 return 한다
 - ▶ 주의: 컴포넌트의 이름은 PascalCase로 작성
- ▶ [실습] 컴포넌트 만들기
 - ▶ 환영 메시지를 출력하는 Welcome 컴포넌트를 만들어 페이지에 렌더링 해 보시다

```
class Welcome extends React.Component {  
  constructor() {  
    // 생성자  
    super();  
  }  
  
  render() {  
    // Rendering...  
    return (  
      <div>  
        // JSX  
      </div>);  
    }  
}
```

How React Works?

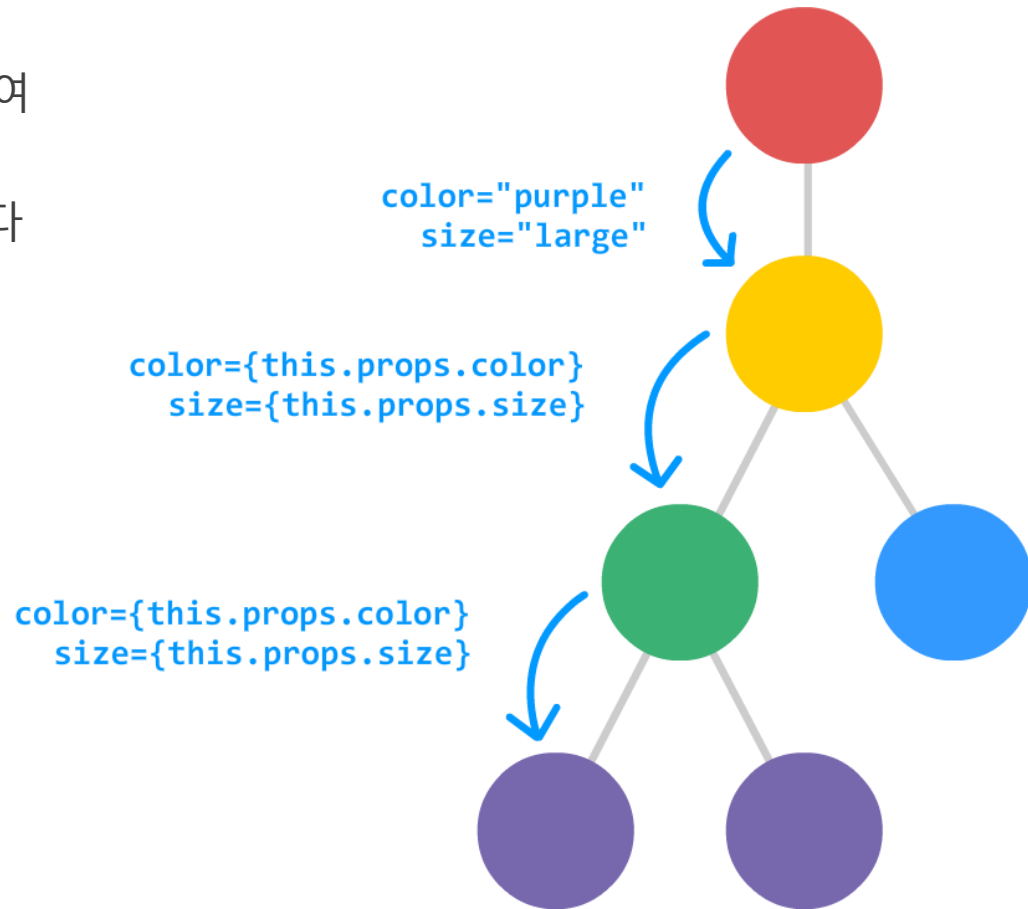
: React Developer Tools

- ▶ React 컴포넌트는 실제 html 엘리먼트로 변환되어 Real DOM에 Rendering된다
 - ▶ 개발 도중, 실제 html 엘리먼트가 아닌 리액트의 컴포넌트를 보고자 한다면 React Developer Tools의 도움을 받을 수 있다.
- ▶ React Developer Tools
 - ▶ <https://chrome.google.com/webstore/>에서 React Developer Tools 검색 후 설치
 - ▶ React가 적용된 사이트에서 크롬 개발자 도구 > Components 탭에서 확인 가능
 - ▶ 컴포넌트의 구성 정보 확인
 - ▶ props, state 등의 정보 확인 및 변경 가능
- ▶ [실습] React Developer Tools
 - ▶ 개발 도구를 설치하고 직접 작성한 컴포넌트의 정보를 확인, 변경해 보시다.

How React Works?

: props

- ▶ 사용자 컴포넌트를 생성할 때 속성(attribute)를 부여하면 자식 컴포넌트에 데이터를 전달해 줄 수 있다.
 - ▶ 전달된 props는 컴포넌트 객체의 props 속성을 이용하여 조회할 수 있다
 - ▶ 문자열은 ""로, 그 이외의 데이터는 객체 {}로 전달한다
 - ▶ props는 읽기 전용(Read-Only)으로 컴포넌트 객체 내부에서 임의로 변경할 수 없다
- ▶ [실습] props의 확인
 - ▶ Welcome 컴포넌트에 message prop을 전달하여 렌더링을 해 보시다.
 - ▶ props 속성을 출력해 보고 어떤 데이터가 컴포넌트로 전달되는지 확인해 보시다.

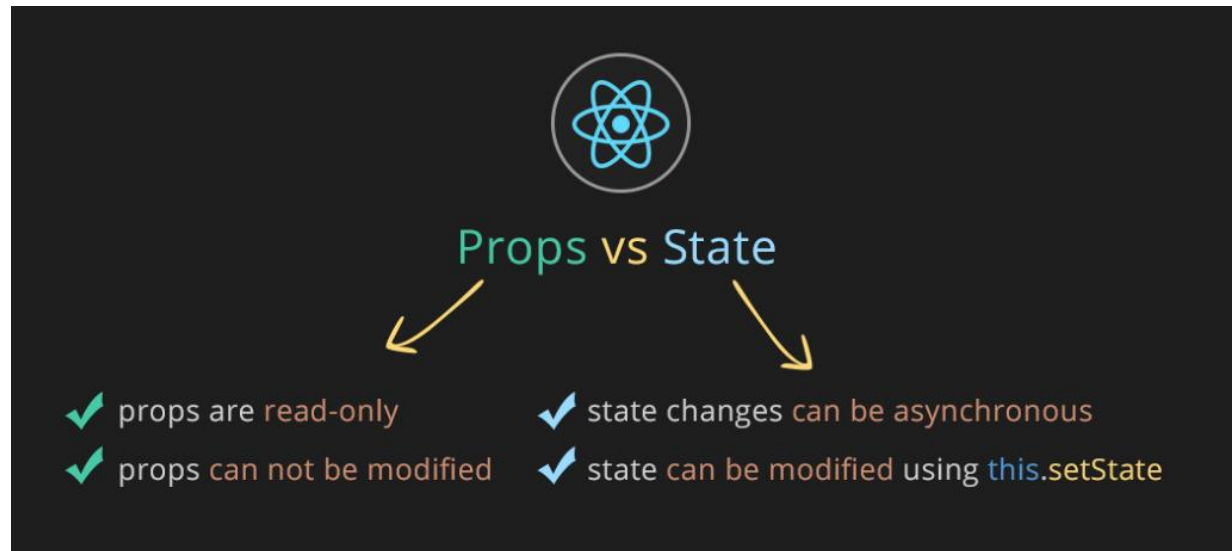


How React Works?

: state

- ▶ props가 외부로부터 주어지는 데이터라면 state는 컴포넌트 내부에서 정의되고 활용되는 데이터
 - ▶ props가 외부로부터 주어진 읽기 전용 데이터인 것에 비해 state는 내부에서 자유롭게 변경할 수 있다.
 - ▶ * 리액트가 약속하는 것:
props나 state가 변경되면 컴포넌트의 render() 함수가 호출될 것!
- ▶ [실습] state 선언하고 사용하기
 - ▶ 컴포넌트 생성자에 color state를 생성하여 style에 반영해 보시다.

```
constructor(props) {  
  // 생성자  
  super(props);  
  this.state = { color: "black" };  
}
```



How React Works?

: event

▶ 컴포넌트에 이벤트를 연결하려면(Class 컴포넌트)

- (1) 이벤트 메서드(함수)를 선언하고
- (2) 메서드에 `this`를 바인드하고
- (3) `render()` 메서드에서 출력하는 태그와 이벤트 속성에 메서드를 입력하여 이벤트를 연결

```
// ...
constructor(props) {
  super(props);
  this.click = this.click.bind(this);
}
// ...
render() {
  return (
    <button onClick={this.click}>
      Button Name
    </button>
  )
}
```

React에서는 html과 달리 이벤트명에 camelCase를 사용하므로 주의하여 명명

<https://ko.reactjs.org/docs/handling-events.html>

How React Works?

: style

▶ style을 지정하고자 할 때

- (1) style로 사용할 객체를 생성하고
- (2) 컴포넌트의 style 속성에 객체를 할당

```
const style= {  
  color: "yellow",  
  backgroundColor: "black"  
}  
  
return (  
  <div>  
    <h3 style={style}>Styled</h3>  
  </div>);
```



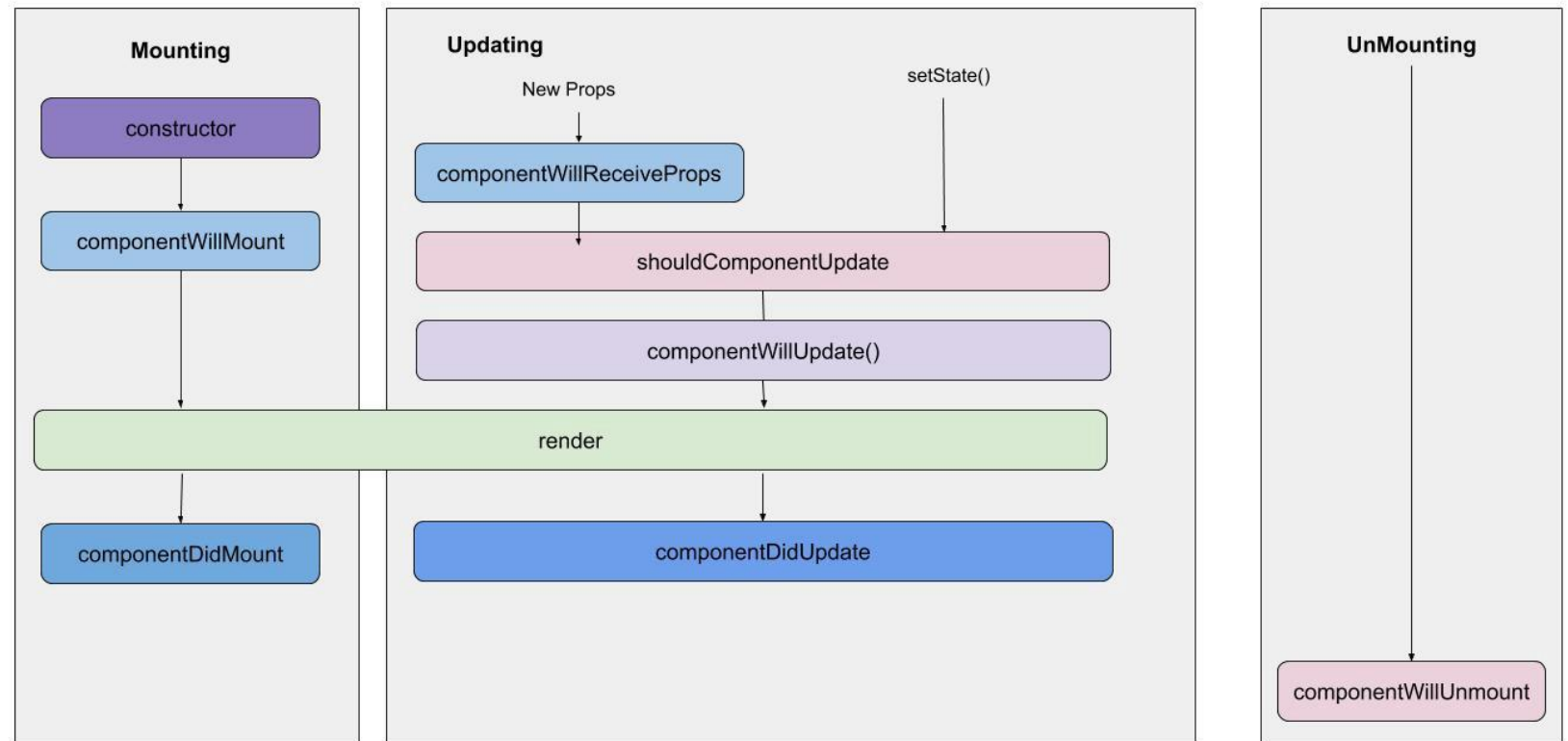
- React에서는 html과 달리 style 객체에 속성명을 camelCase로 부여
- 숫자를 입력할 때는 단위를 입력하지 않아도 됨

How React Works?

: Component Lifecycle

- ▶ 모든 컴포넌트들은 그 자신만의 생애 주기(Lifecycle)을 가짐
- ▶ Old Lifecycle
 - ▶ [실습] Welcome 컴포넌트에 Old Lifecycle을 구현하고 콘솔 창을 통해 호출의 순서를 살펴 봅니다

React 16.0 Life Cycle



How React Works?

: Component Lifecycle

- ▶ 모든 컴포넌트들은 그 자신만의 생애 주기(Lifecycle)을 가짐
- ▶ New Lifecycle
 - ▶ [주의] 구 라이프사이클도 사용 가능하나 구 라이프사이클과 신규 라이프사이클을 함께 사용하면 안됩니다!

React 16.4 Life Cycle



How React Works?

: Component Lifecycle

▶ Mount 단계

▶ constructor

- ▶ 컴포넌트가 만들어질 때 가장 먼저 실행


▶ getDerivedStateFromProps (should be static)

- ▶ 다른 메서드와 달리 static 이어야 하며 이 내부에서는 this를 조회할 수 없음
- ▶ props로 받아온 것을 state에 넣어주고 싶을 때 사용

▶ render

- ▶ 컴포넌트를 렌더링

▶ componentDidMount




```
static getDerivedStateFromProps(nextProps, prevState) {  
  // props로 받아온 것을 state에 넣어주고 싶을 때 사용  
  if (nextProps.color !== prevState.color) {  
    return { color: nextProps.color }  
  }  
  return null;  
}
```

How React Works?


: Component Lifecycle

▶ Update 단계

- ▶ `getDerivedStateFromProps`
- ▶ `shouldComponentUpdate`
 - ▶ 컴포넌트를 업데이트 할 것인지 말 것인지를 결정
- ▶ `render`
- ▶ `getSnapshotBeforeUpdate`
 - ▶ 컴포넌트에 변화가 일어나기 전의 DOM 상태를 가져와서 특정 값을 반환하면 `componentDidUpdate` 메서드로 값이 전달된다.
- ▶ `componentDidUpdate`
 - ▶ 리렌더링을 마치고 화면에 원하는 변화가 모두 반영된 후 호출되는 메서드



```
shouldComponentUpdate(nextProps, nextState) {  
  // 컴포넌트가 리렌더링 할지 말지를 결정하는 메서드  
  console.log("shouldComponentUpdate calls",  
              nextProps, nextState);  
  return false; // true를 반환하면 컴포넌트를 다시 렌더링  
}
```



```
componentDidUpdate(prevProps, prevState, snapshot) {  
  if (snapshot) {  
    // snapshot -> 업데이트 되기 전의 상태  
  }  
}
```

How React Works?

: Component Lifecycle

▶ Unmount 단계

▶ `componentWillUnmount`

- ▶ 컴포넌트가 화면에서 사라지기 직전에 호출
- ▶ 주요 처리 작업
 - ▶ 직접 DOM에 등록했던 이벤트 제거
 - ▶ 등록된 타이머가 있다면 해제
 - ▶ 사용한 외부 라이브러리의 `dispose` 기능이 있다면 호출하여 사용한 자원을 해제

▶ [실습] Lifecycle

- ▶ 새로운 Lifecycle 메서드를 구현하고 콘솔을 이용하여 호출 순서 및 상태 변화를 확인해 보시다.

Create React App

React 작업 환경 준비

: Create React App (aka. CRA)

▶ Requirements

- ▶ Node.js
- ▶ npx, yarn
- ▶ Visual Studio Code (or Other Text Editor)
- ▶ React Developer Tools (구글 크롬 확장 프로그램)

▶ React 앱 생성

```
npx create-react-app react-practice  
cd react-practice  
yarn start
```



Edit `src/App.js` and save to reload

[Learn React](#)

React 작업 환경 준비

: 프로젝트 살펴보기

▶ package.json

▶ 프로젝트 일반 정보

```
{  
  "name": "react-practice",  
  "version": "0.1.0",  
  "private": true,  
  ...  
}
```

▶ 의존성 정보

```
...  
  "dependencies": {  
    ...  
    "react": "^17.0.1",  
    "react-dom": "^17.0.1",  
    "react-scripts": "4.0.2",  
    ...  
  },  
  ...  
}
```

개발 버전의 의존성은 devDependencies

React 작업 환경 준비

: 프로젝트 살펴보기

- ▶ package.json

- ▶ scripts

- ▶ npm or yarn 명령어로 스크립트 수행 가능

```
"scripts": {  
  "start": "react-scripts start",  
  "build": "react-scripts build",  
  "test": "react-scripts test",  
  "eject": "react-scripts eject"  
},
```

- ▶ public 폴더: 실제 웹으로 서비스되는 디렉터리

```
<!-- index.html -->  
<noscript>You need to enable JavaScript to run this app.</noscript>  
<div id="root"></div>
```

컴파일된 js 소스가 이곳에 연결된다

React 작업 환경 준비

: 프로젝트 살펴보기

▶ src 폴더

- ▶ React 구동을 위한 스크립트 파일이 이곳에 위치

App.js



```
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```

React 작업 환경 준비

: 프로젝트 살펴보기

▶ src 폴더

- ▶ App.js, index.js, index.html의 관계를 알아보자

index.html

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

App.js

```
...  
  
function App() {  
  return (  
    <div className="App">  
      ...  
    </div>  
  );  
}  
  
export default App;
```

Compile

index.js

```
ReactDOM.render(  
  <React.StrictMode>  
    <App />  
  </React.StrictMode>,  
  document.getElementById('root')  
>);
```



React Component

- ▶ 컴포넌트 : 독립적인 기능을 수행하는 소프트웨어 모듈
 - ▶ 컴포넌트는 화면을 구성할 수 있는 블록(화면의 영역)을 의미
 - ▶ 화면을 빠르게 구조화 하여 일괄적인 패턴으로 개발할 수 있음
 - ▶ 코드의 재사용에 유리
- ▶ 첫 번째 컴포넌트 만들기

```
import React from 'react';  
  
function Item() {  
  return (  
    <h3>My Item: Notebook</h3>  
  )  
}  
  
export default Item;
```

./components/Item.js

React Component

- ▶ 만들어진 컴포넌트를 App 컴포넌트에 자식으로 추가

```
import Item from './components/Item';  
  
function App() {  
  return (  
    <div className="App">  
      <Item />  
    </div>  
  )  
}  
  
export default App;
```

App.js

./components/Item.js

JSX

: JavaScript XML

- ▶ JavaScript와 HTML을 조합한 문법
 - ▶ <https://ko.reactjs.org/docs/introducing-jsx.html>
 - ▶ JavaScript 코드 내에 UI를 구성하는 태그를 포함시켜 코딩하는 방식
- ▶ 규칙 1
 - ▶ JSX 내의 주석은 `{/* ~ */}`으로 표시한다
- ▶ 규칙 2
 - ▶ JSX 내에 자바스크립트 표현식을 포함할 수 있다.
 - ▶ `{ JavaScript 표현식 }`

JSX

: JavaScript XML

▶ 규칙 3: 속성의 정의

- ▶ 속성에 따옴표를 이용하여 문자열 리터럴을 정의할 수 있다

```
const element = <div tabIndex="0"></div>;
```

- ▶ 중괄호를 사용하여 속성에 자바스크립트 표현식을 삽입할 수 있다.

```
const element = <img src={user.avatarUrl}></img>;
```

▶ 주의:

- ▶ 자바스크립트 표현식을 속성에 삽입할 때, 따옴표를 입력하면 안된다
- ▶ JSX에서 속성명을 사용할 때는 camelCase로 명명하도록 한다
 - ▶ class -> className, tabIndex -> tabIndex 등

JSX

: JavaScript XML

▶ 규칙 4: 자식 컴포넌트

- ▶ 컴포넌트는 자식을 포함할 수 있다.
- ▶ 자식 태그(혹은 컴포넌트)가 없다면 단독 태그 형태로 닫아주는 것을 권장한다

```
const element = <img src={user.avatarUrl} />;
```

▶ 규칙 5: 최상위 태그

- ▶ JSX는 반드시 1개의 최상위 태그가 있어야 한다
- ▶ 불필요한 HTML 요소 생성을 피하기 위해서는 Fragment를 사용할 수 있다.
 - ▶ `React.Fragment`
 - ▶ `<></>`

Props

: 컴포넌트에 값 전달하기

- ▶ props: 부모 컴포넌트에서 자식 컴포넌트로 전달하는 데이터

```
import Item from './components/Item';

function App() {
  return (
    <div className="App">
      <Item name="notebook" />
      <Item name="smartphone" />
      <Item name="tablet" />
    </div>
  )
}

export default App;
```

App.js

```
function Item(props) {
  return (
    <h3>My Item: { props.name }</h3>
  )
}
```

./components/Item.js

```
function Item({name}) {
  return (
    <h3>My Item: { name }</h3>
  )
}
```

구조 분해 할당 방식 응용

주의! 문자열 이외의 데이터를 전달할 때는 { }로 감싸야 합니다!

조건부 렌더링

- ▶ 전달 받은 특정 props 값으로 다른 렌더링 결과를 얻어내고자 할 때 사용
 - ▶ 3항 연산자의 이용

```
function App() {  
  return (  
    <div className="App">  
      <Item name="notebook" checked={true} />  
      ...  
    </div>  
  )  
}
```

App.js

```
function Item({name, checked}) {  
  return (  
    <h3>  
      { checked ? <b>*</b>: null }  
      My Item: { name }  
    </h3>  
  )  
}
```

./components/Item.js

조건부 렌더링

- ▶ 전달 받은 특정 props 값으로 다른 렌더링 결과를 얻어내고자 할 때 사용
 - ▶ 논리 연산자 이용

```
function App() {  
  return (  
    <div className="App">  
      <Item name="notebook" checked={true} />  
      ...  
    </div>  
  )  
}
```

App.js

```
function Item({name, checked}) {  
  return (  
    <h3>  
      { checked && <b>*</b> }  
      My Item: { name }  
    </h3>  
  )  
}
```

./components/Item.js

배열 렌더링

: 다수의 컴포넌트 렌더링

▶ 데이터를 저장할 배열을 선언

```
const items = [  
  {  
    id: 1,  
    name: "notebook",  
    checked: true  
  },  
  {  
    id: 2,  
    name: "smartphone",  
    checked: false  
  },  
  {  
    id: 3,  
    name: "tablet",  
    checked: true  
  },  
];
```

App.js

▶ 배열을 순회하면서 렌더링

```
function App() {  
  return (  
    <div className="App">  
      {  
        items.map(item => (  
          <Item key={item.id}  
            name={item.name}  
            checked={item.checked} />  
        ))  
      }  
    </div>  
  )  
}
```

App.js

컴포넌트를 반복 렌더링할 때는
key 프로퍼티에 유일 식별자를 부여

prop-types

- ▶ prop 값이 컴포넌트에 제대로 전달되지 않으면 정상적으로 작동하지 않을 수 있음
 - ▶ props를 검사하는 방법이 필요

```
npm install prop-types # npm
yarn add prop-types # yarn
```

- ▶ 컴포넌트 소스 파일에 prop-types 정보를 추가

```
...
import PropTypes from 'prop-types';

...
Item.propTypes = {
  name: PropTypes.string.isRequired,
  checked: PropTypes.bool.isRequired
}
...
./components/Item.js
```

prop-types

- ▶ prop 값이 컴포넌트에 제대로 전달되지 않으면 정상적으로 작동하지 않을 수 있음
 - ▶ 컴포넌트 소스 파일에 defaultProps 속성을 주어 기본 props 값을 설정할 수 있다.

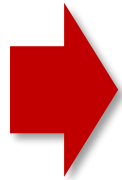
```
...  
import PropTypes from 'prop-types';  
  
...  
Item.defaultProps = {  
  name: "Unnamed",  
  checked: false  
}  
...  
./components/Item.js
```

Props

: 컴포넌트에 값 전달하기

- ▶ props: 부모 컴포넌트에서 자식 컴포넌트로 전달하는 데이터
 - ▶ in Class Component (Example)

```
...  
<PropsClass prop_str="string value"  
  prop_num={ 10 }  
  prop_false={ false }  
  prop_true  
  prop_obj={{  
    name: "홍길동",  
    age: 28  
  }} />  
...  
./App.props.jsx
```



```
...  
let { prop_str,  
    prop_num,  
    prop_false,  
    prop_true,  
    prop_obj } = this.props;  
...  
./components/PropsClass.jsx
```

전달된 props는 `this.props`로 객체에 전달된다

- ▶ 문자열은 `"`로 전달
- ▶ 그 이외의 값은 `{}`로 전달
- ▶ 객체를 전달하고자 할 때는 `{{ ... 객체 ... }}`

Props

: 컴포넌트에 값 전달하기

- ▶ props: 부모 컴포넌트에서 자식 컴포넌트로 전달하는 데이터
 - ▶ in Function Component (Example)

```
...  
<PropsFunc prop_str="string value"  
  prop_num={ 10 }  
  prop_false={ false }  
  prop_true  
  prop_obj={{  
    name: "홍길동",  
    age: 28  
  }} />  
...  
./App.props.jsx
```



```
...  
let { prop_str,  
    prop_num,  
    prop_false,  
    prop_true,  
    prop_obj } = this.props;  
...  
./components/PropsFunc.jsx
```

전달된 props는 `this.props`로 객체에 전달된다

- ▶ 문자열은 `"`로 전달
- ▶ 그 이외의 값은 `{}`로 전달
- ▶ 객체를 전달하고자 할 때는 `{{ ... 객체 ... }}`

Props

: 컴포넌트에 값 전달하기

- ▶ props: 부모 컴포넌트에서 자식 컴포넌트로 전달하는 데이터
 - ▶ Boolean 데이터의 전달
 - ▶ props 변수를 Boolean 형으로 자식 컴포넌트에 전달할 경우, true or false 하나를 할당
 - ▶ props 변수를 선언한 후, 값을 할당하지 않고 전달하면 기본값으로 true가 할당

```
...  
<PropsFunc  
  prop_false={ false }  
  prop_true  
>  
...  
./App.props.jsx
```

=

```
...  
<PropsFunc  
  prop_false={ false }  
  prop_true={ true }  
>  
...  
./App.props.jsx
```


Props

: 컴포넌트에 값 전달하기

- ▶ props: 부모 컴포넌트에서 자식 컴포넌트로 전달하는 데이터
 - ▶ 객체 데이터의 전달
 - ▶ {{ ... 객체 ... }} 형태로 전달되어야 하며
 - ▶ 객체 내부 변수들의 자료형을 선언할 때는 shape라는 유형을 사용한다.

```
...  
<PropsFunc  
  prop_obj={{  
    name: "홍길동",  
    age: 28  
  }}  
/>  
...  
./App.props.jsx
```

```
...  
PropsClass.propTypes = {  
  prop_obj: datatype.shape({  
    name: datatype.string,  
    age: datatype.number  
  })  
}  
...  
./components/PropsClass.jsx
```

Props

: 컴포넌트에 값 전달하기

- ▶ props: 부모 컴포넌트에서 자식 컴포넌트로 전달하는 데이터
 - ▶ 자식 컴포넌트에 node 전달하기
 - ▶ props는 하위 컴포넌트 태그 내에 작성된 HTML 노드들도 전달할 수 있다 (children)

```
...  
<PropsClass>  
  <h3>Children</h3>  
  <p>자식 컴포넌트</p>  
</PropsClass>  
...
```

./App.props.jsx

```
...  
return (  
  <div>  
    <h2>Props from Parent</h2>  
    { this.props.children }  
  </div>  
)  
...
```

./components/PropsClass.jsx

State

: 컴포넌트 내부에서 변경되는 값 관리하기

- ▶ state : 하나의 컴포넌트 내에서 전역 변수로 활용되는 데이터
 - ▶ state는 컴포넌트 클래스의 constructor에서 초깃값을 설정한다
 - ▶ state 변수에 접근하기 위해서는 **this.state.변수명** 으로 표기한다.

```
...
function App() {
  return (
    <div>
      <h2>State in Component</h2>
      <ReactState reactString={"react"} />
    </div>
  )
}
...
./App.state.jsx
```

```
...
constructor(props) {
  super(props);
  this.state = {
    stateString:
      this.props.reactString,
    stateNumber: 2021,
  };
}
...
./components/ReactState.jsx
```

```
...
<li>{ this.state.stateString }</li>
<li>{ this.state.stateNumber }</li>
...
./components/ReactState.jsx
```

State

: 컴포넌트 내부에서 변경되는 값 관리하기

- ▶ state를 직접 변경하면 생기는 문제
 - ▶ state를 직접 변경하게 되면 값은 변경되지만
 - ▶ render() 함수가 호출되지 않는다
- ▶ [실습] ReactState.jsx를 변경
 - 버튼을 생성하고 이벤트를 연결
 - state를 직접 변경하는 코드를 만들어 봅니다.
 - 화면에 변경된 state가 잘 rendering 되는지 확인합니다.

```
...
stateChange = (flag) => {
  if (flag === "direct") {
    this.state.stateString = "리액트";
    this.state.stateNumber += 1;
  }
}
...
./components/ReactState.jsx
```

```
...
<button onClick={ e => this.stateChange("direct", e) }>
  State 직접 변경
</button>
...
./components/ReactState.jsx
```

State

: 컴포넌트 내부에서 변경되는 값 관리하기

▶ state를 직접 변경하면

- ▶ render() 함수가 호출되지 않아 변경되는 state가 화면에 반영되지 않음
- ▶ state 변경을 위해 setState() 함수를 이용하면 render() 함수가 호출되어 변경된 값을 화면에 반영할 수 있다.

▶ [실습]

- 버튼을 생성하고 이벤트를 연결
- setState() 함수를 이용하여 state를 변경
- 변경된 state가 화면에 잘 rendering 되는지 확인합니다.

```
...
stateChange = (flag) => {
  if (flag === "setState") {
    this.setState({
      stateString: "리액트",
      stateNumber: this.state.stateNumber + 1
    });
  }
}
...
```

./components/ReactState.jsx

```
...
<button onClick={ e => this.stateChange("setState", e) }>
  setState 변경
</button>
...
```

./components/ReactState.jsx

State

: 컴포넌트 내부에서 변경되는 값 관리하기

▶ state를 직접 변경하면

- ▶ render() 함수가 호출되지 않아 변경되는 state가 화면에 반영되지 않음
- ▶ 이때, forceUpdate() 함수를 호출하면
강제로 render() 함수를 호출하여
변경된 state를 화면에 반영할 수 있다.

▶ [실습]

- 버튼을 생성하고 이벤트를 연결
- forceUpdate() 함수를 호출
- 변경된 state가 화면에 잘 rendering 되는지 확인합니다.

```
...  
stateChange = (flag) => {  
  if (flag === "forceUpdate") {  
    this.forceUpdate();  
  }  
}  
...  
./components/ReactState.jsx
```

```
...  
<button onClick={ e => this.stateChange("forceUpdate", e) }>  
  state 강제 변경  
</button>  
...  
./components/ReactState.jsx
```

Functional Components

함수형 컴포넌트

- ▶ 함수형 컴포넌트(Function Component)는 클래스형 컴포넌트와 달리
 - ▶ state가 없고
 - ▶ 생명주기 함수를 사용할 수 없다
- ▶ 상위 컴포넌트에서 props와 context를 파라미터로 전달받아 사용
- ▶ render 함수가 없기 때문에 return만 사용하여 화면을 그린다.

```
...  
function App() {  
  return (  
    <div className="App">  
      <h2>Function Component</h2>  
      <UseStateComp contents="[이것은 함수형  
컴포넌트입니다]" />  
    </div>  
  );  
}  
...  
./App.function.jsx
```

```
...  
function UseStateComp(props) {  
  let { contents } = props;  
  return (  
    <h2>{ content }</h2>  
  )  
}  
export default UseStateComp;  
...  
./components/UseStateComp.jsx
```


Hook

: useState

- ▶ 함수형 컴포넌트에서 상태 관리, 생명 주기 관리 등 클래스 컴포넌트에서 할 수 있는 방법을 제공하는 기능
- ▶ useState
 - ▶ 가장 기본적인 Hook
 - ▶ 함수형 컴포넌트에서도 가변적인 상태를 지닐 수 있게 해 주는 hook
- ▶ useState 혹은 import

```
import { useState } from 'react';
```

- ▶ useState 선언하기

```
const [count, setCount] = useState(0);
```

상태 변수명

상태 설정 함수

상태 초깃값

<https://ko.reactjs.org/docs/hooks-state.html>

Hook

: useState

▶ state 값 변경하기

- ▶ useState로 생성한 상태 설정 함수로 state 값을 변경할 수 있다.

```
...  
return (  
  <div>  
    <h2>{ contents }</h2>  
    <p>현재 카운트 값은 {count} 입니다.</p>  
    <button onClick={() => setCount(count + 1)}>+1</button>  
    <button onClick={() => setCount(count - 1)}>-1</button>  
  </div>  
)  
...
```

./components/UseStateComp.jsx

- ▶ 하나의 useState 함수는 하나의 상태 값만 관리할 수 있다.
 - ▶ 여러 개의 상태를 관리하고자 하면 useState를 여러 번 사용해야 한다.

Hook

: useEffect

▶ useEffect

- ▶ 리액트 컴포넌트가 렌더링 될 때마다 특정 작업을 수행하도록 설정하는 Hook
- ▶ 클래스 컴포넌트의 componentDidMount와 componentDidUpdate를 합친 형태

▶ useEffect 혹은 import

```
import { useEffect } from 'react';
```

- ▶ useEffect 호출 1: 렌더링 될 때마다 호출하고자 할 때
 - ▶ componentDidMount + componentDidUpdate

```
useEffect(() => {  
  console.log("useEffect called");  
});
```

Hook

: useEffect

▶ useEffect

- ▶ useEffect 호출 2: 맨 처음 렌더링될 때만 실행하고자 할 때
 - ▶ 두 번째 파라미터로 빈 배열([])을 전달한다.

```
useEffect(() => {  
  console.log("useEffect called");  
}, []);
```

- ▶ useEffect 호출 3: 특정 값이 변경될 때만 호출하고 싶을 때
 - ▶ 두 번째 파라미터로 전달되는 배열 안에 검사하고 싶은 값을 넣는다

```
useEffect(() => {  
  console.log("count:", count);  
}, [count]);
```



Dependency

Hook

: useEffect

▶ useEffect

- ▶ useEffect는 기본적으로 렌더링되고 난 직후마다 실행되며 두 번째 파라미터의 배열에 무엇을 넣는지에 따라 실행 조건이 달라진다.
- ▶ 컴포넌트가 언마운트되기 전이나 업데이트 직전에 특정 작업을 수행하고 싶다면 useEffect에서 뒷정리 함수(Cleanup Function)을 호출해 주어야 한다

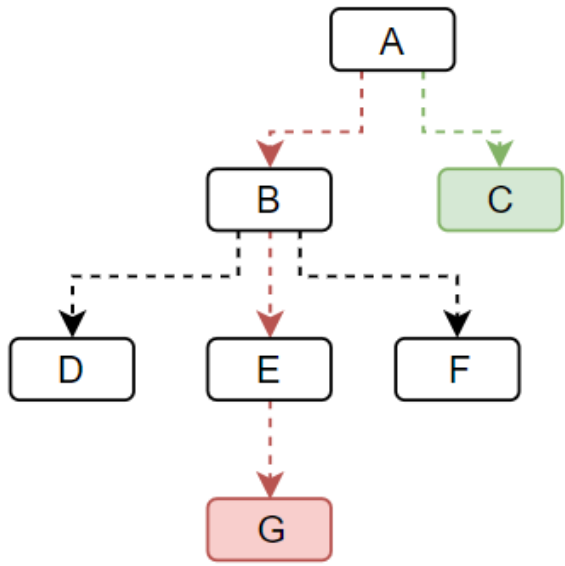
```
useEffect(() => {  
  console.log("useEffect");  
  console.log("count:", count);  
  return () => {  
    // cleanup 함수  
    console.log("cleanup");  
    console.log(count);  
  }  
});
```

- ▶ 오직 언마운트 될 때에만 뒷정리 함수를 호출하고 싶다면 useEffect 함수의 두 번째 파라미터에 비어 있는 배열을 넣으면 된다.

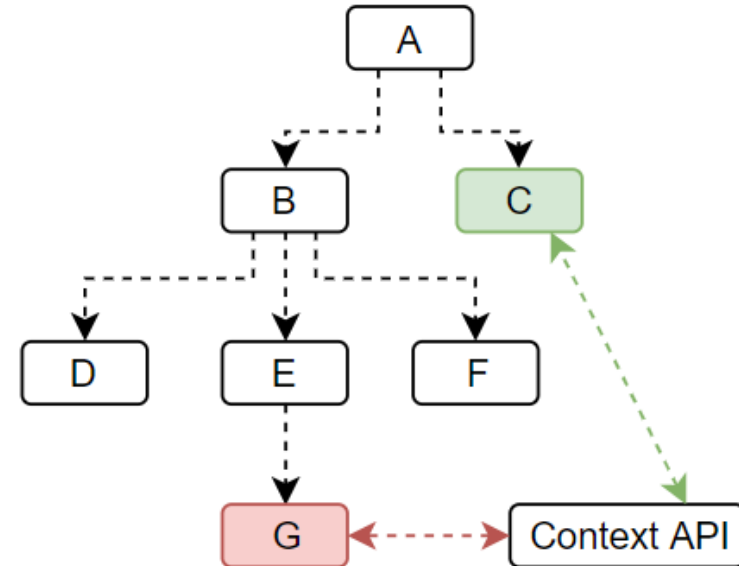
고급 주제

Context API

- ▶ 일반적인 React 응용프로그램의 데이터는 위에서 아래로 props로 전달
 - ▶ 응용프로그램 내 여러 컴포넌트에게 전해줘야 하는 데이터의 경우 번거로울 수 있다
(또한 의미 없는 데이터 릴레이 코드가 발생할 수 있다)
- ▶ Context는 데이터의 공급자와 소비자를 정의하고 데이터가 필요한 컴포넌트만 사용할 수 있게 구현할 수 있다.



컴포넌트를 통한 상태 공유



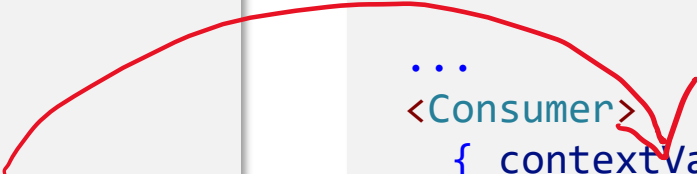
Context API를 통한 상태 공유

Context API

- ▶ Context API의 Provider로 데이터 제공하기
 - ▶ createContext 함수로 공급자(Provider)와 소비자(Consumer)를 받아 사용
 - ▶ 하위 컴포넌트에서 소비자를 사용할 수 있도록 Export
 - ▶ 자식 컴포넌트를 Provider 태그로 감싸고 전달할 데이터를 value 값으로 할당
- ▶ Consumer로 데이터 소비하기
 - ▶ Export한 Consumer를 Import
 - ▶ Consumer 태그로 출력할 Element를 감싸고 Provider에 할당한 데이터를 받아온다

```
...  
const { Provider, Consumer } =  
  React.createContext();  
export { Consumer }  
  
...  
<Provider value="Using Context API">  
  <Children />  
</Provider>  
...  
./components/ContextApi.jsx
```

```
...  
import { Consumer } from './ContextApi';  
  
...  
<Consumer>  
  { contextValue =>  
    <h3>`contextValue: ${contextValue}`</h3>  
  }  
</Consumer>  
...  
./components/ContextChildren2.jsx
```



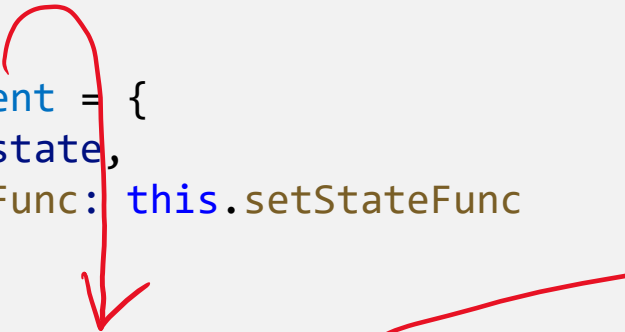
Context API

: 컨텍스트로 부모 데이터 변경

- ▶ props는 데이터가 부모에서 자식 컴포넌트의 단방향으로만 이동할 수 있다.
컨텍스트를 사용하면 자식 컴포넌트에서 부모 컴포넌트의 데이터를 변경할 수 있다.

```
...
setStateFunc(value) {
  this.setState({ name: value });
}

...
const content = {
  ...this.state,
  setStateFunc: this.setStateFunc
}
return (
  <Provider value={content}>
    <Children />
  </Provider>
)
...
./components/ContextApi.jsx
```



```
...
<Consumer>
  { contextValue =>
    <button onClick={ e => contextValue
      .setStateFunc("Message From Child")}>
      { contextValue.name } button
    </button>
  }
</Consumer>
...
./components/ContextChildren2.jsx
```

Ref의 사용

▶ Ref(Reference: 참조)

- ▶ 일반적으로 React에서 자식 객체를 수정하려면 새로운 props를 전달하여 자식을 다시 렌더링 해야 함
- ▶ 경우에 따라, 직접 자식을 수정해야 하는 경우도 있다.

▶ Ref의 생성

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

    this.inputRef = React.createRef();
  }
...

<input id="id" type="text"
      ref={this.inputRef} />

./components/ReactRef.jsx
```

▶ Ref에 접근

- ▶ render 메서드 내에서 ref가 엘리먼트에 전달되었을 때, 노드를 향한 참조는 ref의 current 속성에 담긴다.

```
...
this.textInput.current.focus();
...
```

Routing

: with react-router-dom

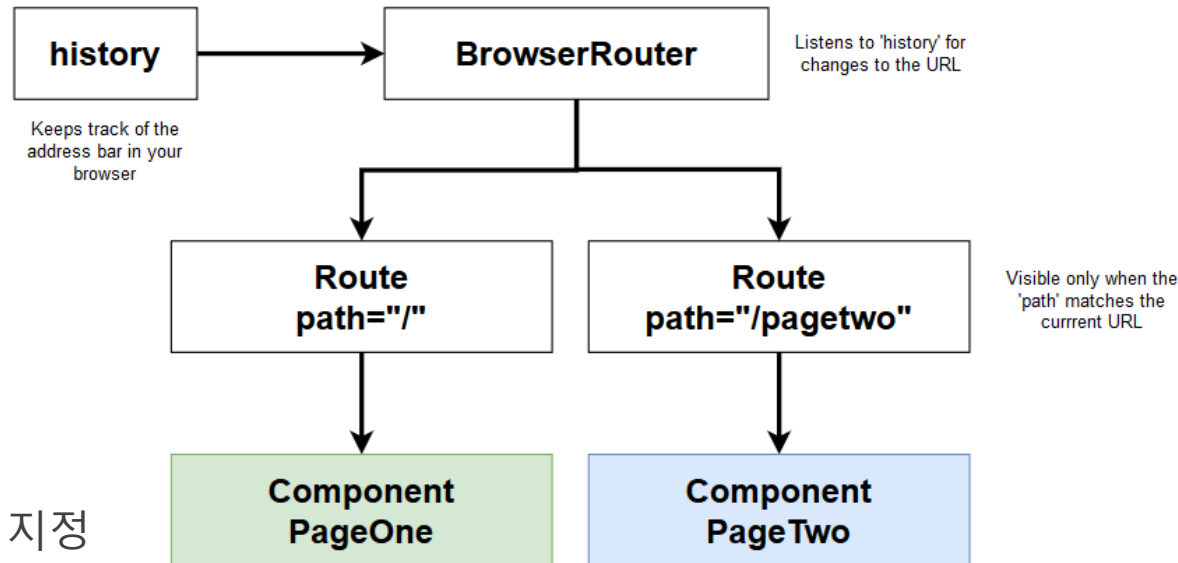
▶ Routing

```
yarn add react-router-dom
```

▶ 제공하는 것들

- ▶ BrowserRouter
- ▶ Route : 호출되는 URL에 따라 이동할 컴포넌트를 지정
- ▶ Link : <a> 태그처럼 클릭하면 url을 호출

- ▶ Route와 Link를 사용하기 위해서는 BrowserRouter 태그로 감싸 사용해야 한다.



```
...  
ReactDOM.render((  
  <BrowserRouter>  
    <App />  
  </BrowserRouter>),  
  document.getElementById('root')  
)  
...  
./index.js
```

Routing

: with react-router-dom

▶ Route

- ▶ 서버에 호출된 url의 path에 따라 연결할 컴포넌트를 정의한다.

```
...  
<div>  
  { /* exact 속성을 빼 보고 테스트 해 보자 */ }  
  <Route exact path="/" component={RouteHomePage} />  
  <Route exact path="/about" component={RouteAboutPage} />  
</div>  
... ./App.routing.jsx
```

- ▶ path 속성: 호출되는 url 경로
- ▶ component 속성: 연결할 컴포넌트를 할당
- ▶ Route 태그의 exact 속성을 제거해 보고 exact 속성의 역할을 이해해 봅니다.

Routing

: with react-router-dom

▶ Link

- ▶ a 태그와 동일한 동작(Client Side Routing)
- ▶ to 속성에 Route 태그에 정의한 path 속성과 같은 값을 입력한다.

```
...  
<div>  
  <Route exact path="/" component={RouteHomePage} />  
  <Route exact path="/about" component={RouteAboutPage} />  
</div>  
...
```

./App.routing.jsx

```
...  
<Link to="/">Link to Home</Link>  
...
```

```
...  
<Link to="/about">Link to About</Link>  
...
```

Routing

: with react-router-dom

- ▶ 라우팅과 상관 없이 항상 표시되어야 하는 영역의 처리
 - ▶ Route 태그 바깥쪽 영역에 배치 시킨다

```
...  
<div>  
  <Route exact path="/" component={RouteHomePage} />  
  <Route exact path="/about" component={RouteAboutPage} />  
</div>  
...                                     ./App.routing.jsx
```