

React

REACT -

목차

01. REACT	4
> 01. react 기초	4
> 02. 리액트 설치	8
> 03. JSX의 기본구조	32
> 04. 화살표 함수 사용법	40
> 04. css 사용	47
> 05. 변수와 scope	51
> 04. 배열	57
> 07. 이벤트 추가하기	69
> 08. hooks useState	75
> 08. 다양한 이벤트	81
전통적인 방식:	89
구조분해 할당 방식:	89
구조 분해 할당 (Destructuring Assignment) 초간단 설명	90
1. 배열 구조 분해	90
기본 사용법	90
기능 추가 설명	90
(1) 필요한 값만 가져오기 (건너뛰기)	90
(2) 기본값 설정 (값이 없을 때)	90
(3) 나머지 값 한 번에 가져오기 (...)	91
2. 객체 구조 분해	91
기본 사용법	91
기능 추가 설명	91
(1) 변수명 변경하기 (새변수: 원래키)	91
(2) 기본값 설정 (키가 없을 때)	91
(3) 중첩 객체 분해	92
💡 구조 분해 할당의 장점	92
🚀 초간단 요약	92
가장 간단한 조건부 렌더링 예제	103
동작 설명:	103
핵심 원리:	103
> 15. 단어장 만들기	108
> 16.	108
> 17. 쉽게 개선선	108
> 18. 마지막 학생생	115
> 14. 다양한 훅 사용법	137
기본 훅	137

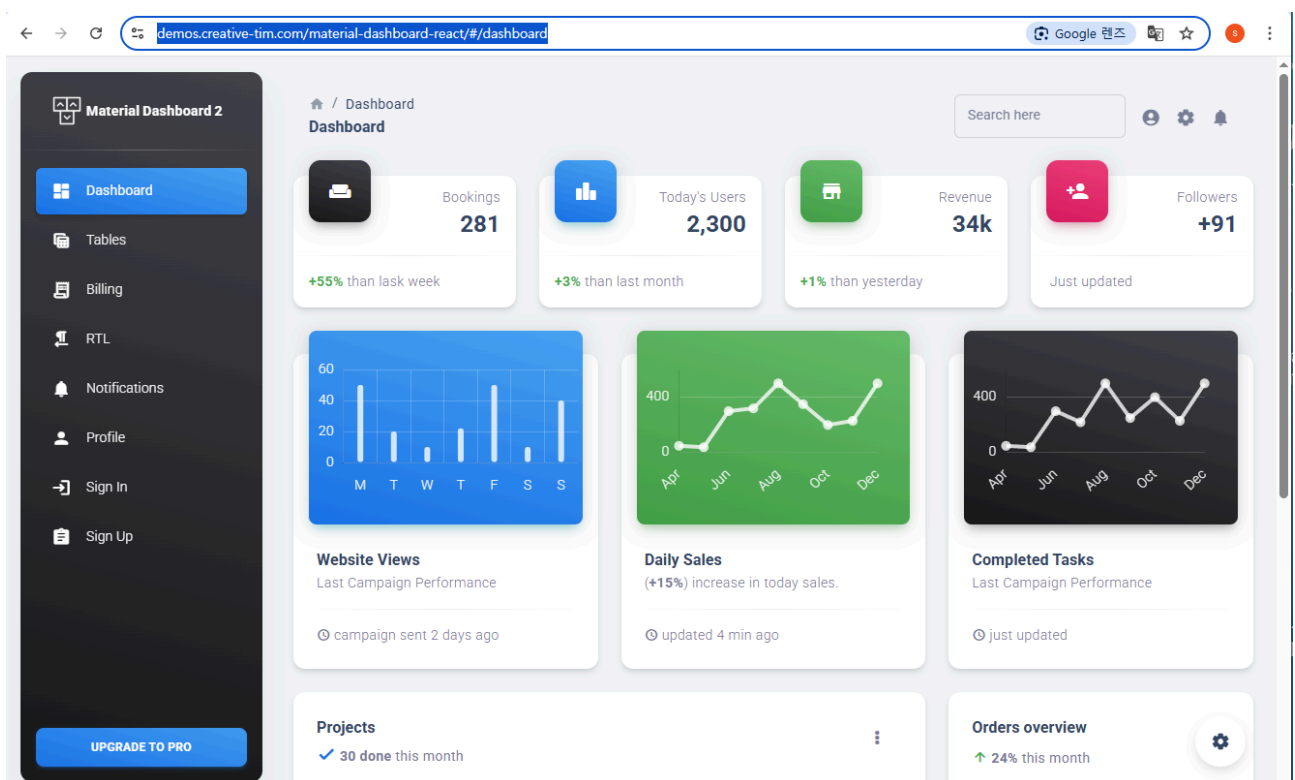
설명	146
기본 문법	147
주요 개념	148
사용 예제	148
1. GET 요청	148
2. POST 요청	148
3. Error Handling	149
비동기/대기 (async/await)	150
결론	151
사용자 정의 훅 (Custom Hook)	153
주요 포인트:	153
> 19.	156
> 20.	156
> 21.	156
> 22.	156
> 23.	157
> 24.	157
> 25.	157
> 26.	157
> 27.	157
> 28.	157
> 29.	157
> 30.	158

01. REACT

> 01. react기초

React는 사용자 인터페이스(UI)를 구축하기 위한 JavaScript 라이브러리

<https://demos.creative-tim.com/material-dashboard-react/#/dashboard>



jQuery는 주로 DOM 조작과 이벤트 처리에 중점을 둔 라이브러리이며, Angular, React, Vue는 더 현대적인 프레임워크나 라이브러리로 복잡한 애플리케이션 구조를 관리하고 상태를 처리하는 데 중점을 둡니다.

Angular, React, Vue와의 비교

- 모듈화: Angular, React, Vue는 컴포넌트 기반 아키텍처를 지원하여 코드의 재사용성과 유지 보수성을 높입니다. jQuery는 주로 직접 DOM 조작에 의존합니다.
- 상태 관리: Angular, React, Vue는 복잡한 상태 관리와 데이터 바인딩을 지원합니다. jQuery는 상태 관리 기능이 부족합니다.

상태 관리: 데이터를 어떻게 저장/관리할지 (참고 정리)

로그인 상태 (로그인O/X)

장바구니 목록 (담긴 상품들)

다크모드 설정 (켜짐/꺼짐)

데이터 바인딩: 데이터와 화면을 어떻게 동기화할지 (자동 연결)

변수 = "안녕" → 화면에도 바로 "안녕" 표시

- 테스트: Angular, React, Vue는 테스트 용이성을 고려하여 설계되었으며, jQuery는 테스트 관련 기능이 제한적입니다.
- 프레임워크/라이브러리의 목표: Angular, React, Vue는 전체 애플리케이션 구조를 관리하기 위해 설계된 반면, jQuery는 특정 기능을 간편하게 구현하는 데 중점을 둡니다.

angular가 1등이었으나 복잡한 사용법으로 react에게 1위 자리를 빼앗겼고, 현재 클래스 기반 복잡한 react 프로그램 개발에서 함수 기반 react 프로그램 방식으로 변경되었다.

더 쉬운 사용 방법으로 뷰가 확장세를 넓혀가고 있는 실정이고, 아직 React은 어려운 프로그램에 속한다. 앞으로 많은 변화가 예상되므로, 깊이 있는 공부 보다는 심플한 형태로 개발하는 것이 낫다.

React는 실질적으로 웹에서 **싱글 페이지 애플리케이션(SPA)**을 만드는 용도로 사용된다. 주로 **동적인 사용자 인터페이스(UI)**를 구축하는 데 활용되며, 웹에서 실행되는 애플리케이션 제작에 사용된다.

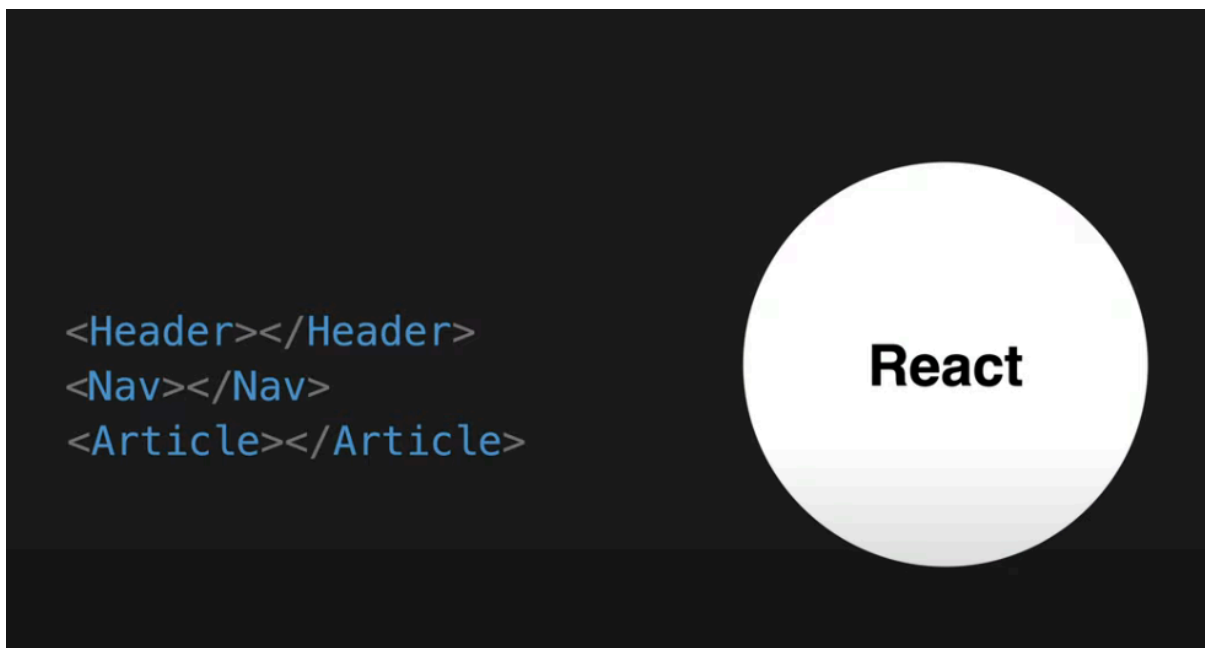
1. React는 웹 애플리케이션을 컴포넌트 기반으로 개발하는 라이브러리

```
<header>
  <h1>
    <a href="/">web</a>
  </h1>
</header>
<nav>
  <ol>
    <li><a href="/read/1">html</a></li>
    <li><a href="/read/2">css</a></li>
    <li><a href="/read/3">javascript</a></li>
  </ol>
</nav>
<article>
  <h2>Welcome</h2>
  Hello, WEB
</article>
```

상위과 같은 html 코드 하위 이미지 같이 매핑 시켜서 컴포넌트로 만든다.

```
<Header></Header>    =    <header>
                             <h1>
                               <a href="/">web</a>
                             </h1>
                             </header>
<Nav></Nav>           =    <nav>
                             <ol>
                               <li><a href="/read/1">ht
                               <li><a href="/read/2">cs
                               <li><a href="/read/3">ja
                             </ol>
                             </nav>
<Article></Article>   =    <article>
                             <h2>Welcome</h2>
                             Hello, WEB
                             </article>
```

리엑을 사용하면 심플하게 컴포넌트화 해서 화면과 프로그램을 만들 수 있다.



class vs function

과거에 리엑트는 클래스 형태로 만들었는데 최근에는 함수 형태로 만들어져 더욱 쉬워져서 더 많이 사용하게 되었다. 1-2년 전 책들만 하더라도 모두 클래스 기반 리엑트로 구현되어 있다.

리엑트 자습서

<https://ko.legacy.reactjs.org/>

리엑트 에디터

<https://stackblitz.com/edit/react-ldxeqc?file=src%2FApp.js>

> 02. 리엑트 설치

✅ npm과 npx는 ? 🚀

◆ 1. npm (Node Package Manager)

👉 패키지를 설치하고 관리하는 도구
(Node.js와 함께 제공됨)

✅ 하는 일:

- 패키지(라이브러리) 설치, 업데이트, 삭제
- 프로젝트의 의존성 관리 (**package.json**)
- 전역(**-g**) 또는 로컬(**node_modules**)에 패키지 설치

✅ 예제:

```
npm install express      # 프로젝트에 express 설치
npm install -g nodemon   # 전역(Global)으로 nodemon 설치
npm update react         # 설치된 react 패키지 업데이트
npm uninstall axios      # axios 패키지 제거
```

♦ 2. npx (Node Package Executor)

👉 설치 없이 패키지를 실행하는 도구
(npm 5.2.0부터 기본 포함됨)

✅ 하는 일:

- 패키지를 설치하지 않고 바로 실행
- 특정 버전의 패키지를 다운로드하여 실행
- 프로젝트 내 패키지를 실행 (`node_modules/.bin` 대신 사용)

✅ 예제:

```
npx create-react-app my-app # 설치 없이 React 프로젝트 생성
npx eslint myfile.js        # eslint를 설치 없이 실행
npx cowsay "Hello World!"   # 설치 없이 실행 후 메시지 출력
```

🚀 결론 (차이점 요약)

구분	npm	npx
역할	패키지 설치 및 관리	패키지를 설치 없이 실행

설치 필요 여부 패키지 설치해야 사용 가능 설치 없이 바로 실행 가능

사용 예시

`npm install axios`

`npx create-react-app
my-app`

- ✓ **npm**은 패키지를 설치하고 관리하는 도구
- ✓ **npx**는 설치 없이 패키지를 바로 실행하는 도구

😊 쉽게 말해서

- 📦 **npm** = "패키지 설치"
- ⚡ **npx** = "설치 없이 실행"

리액트에서 **NPM**과 **NPX** 사용 용도

1. **npm**을 사용하는 경우

npm은 주로 패키지 관리와 관련된 작업을 처리합니다. **React** 프로젝트에서는 주로 다음과 같이 사용됩니다:

◆ 패키지 설치

- **React**와 관련된 라이브러리나 도구들을 프로젝트에 설치할 때 사용됩니다.
- 예: **React**, **React DOM**, 기타 필요한 라이브러리들을 설치

`npm install react react-dom` # **React**와 **ReactDOM**을 설치

`npm install axios` # **HTTP** 요청을 위한 **axios** 설치

npm install react-router-dom # React에서 라우팅을 위한 라이브러리 설치

◆ 의존성 관리

- 설치한 패키지는 **node_modules** 폴더에 저장되고, 이 정보는 **package.json**에 기록됩니다. 이 파일을 통해 나중에 다른 개발자가 프로젝트를 사용할 때 동일한 패키지를 설치할 수 있게 됩니다.

npm install # `package.json`에 기록된 모든 패키지 설치

◆ 스크립트 실행

- **npm**을 사용하여 프로젝트 내에서 정의된 스크립트를 실행할 수 있습니다. 예를 들어, **start**, **build** 등의 명령어를 **package.json**의 **scripts** 섹션에 정의하여 실행합니다.

npm start # React 애플리케이션을 실행 (개발 모드)

npm run build # 배포용 빌드 파일을 생성

2. **npx**를 사용하는 경우

npx는 설치 없이 패키지를 바로 실행하는 데 사용됩니다. **React** 프로젝트에서 주로 사용되는 경우는 **create-react-app** 같은 초기 설정을 할 때입니다.

◆ React 프로젝트 생성

- **React** 애플리케이션을 처음 시작할 때 **npx**를 사용하여 ****create-react-app****을 실행합니다. **create-react-app**은 **React** 프로젝트를 쉽게 설정해주는 도구로, 이 도구를 설치하지 않고도 **npx**로 실행할 수 있습니다.

`npx create-react-app my-app` # `create-react-app`을 실행하여 새로운 React 프로젝트 생성

이 명령어는 `create-react-app`을 전역에 설치하지 않고도 바로 사용할 수 있게 해줍니다. (즉, 매번 최신 버전의 `create-react-app`을 실행할 수 있게 해줍니다.)

◆ `npx`로 실행되는 기타 도구들

- `npx`는 다른 많은 도구들도 설치 없이 실행할 수 있게 해줍니다. 예를 들어, `eslint`, `prettier`, `storybook` 등을 설치 없이 실행할 수 있습니다.

`npx eslint myfile.js` # `eslint`를 설치 없이 실행하여 코드 검사

`npx prettier --write .` # `prettier`를 설치 없이 실행하여 코드 포매팅

◆ React 개발 시 `npm`과 `npx` 사용 요약

기능	<code>npm</code> 사용	<code>npx</code> 사용
프로젝트 생성	-	<code>npx create-react-app my-app</code>
라이브러리 설치	<code>npm install react react-dom</code>	-
프로젝트 실행 (개발 모드)	<code>npm start</code>	-
빌드	<code>npm run build</code>	-

패키지 관리 **npm install <패키지>** -

설치 없이 패키지 실행 - **npx <패키지>**

결론

- **npm**: 패키지 설치, 관리, 실행에 사용됩니다. **React** 라이브러리와 도구들을 설치할 때 주로 사용합니다.
- **npx**: 설치 없이 패키지 실행에 사용됩니다. **React** 프로젝트를 시작할 때 **create-react-app**을 실행하거나, 일시적으로 실행할 도구들을 사용할 때 유용합니다.

둘 다 **React** 개발을 할 때 상호 보완적으로 사용됩니다! 😊

npm과 npx 설치하기

node를 설치하면 자동으로 npm과 npx가 자동 설치된다.

nodejs.org

비주얼 스튜디오 코드 터미널에서 아래 이미지 처럼 개발 환경을 만들어 보자

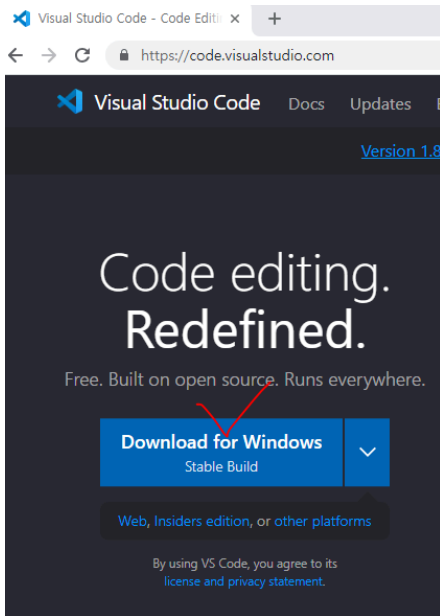
윈도우+R cmd node입력해서 설치 여부 확인

node버전이 출력되면 node가 정상 설치된것이고 npm과 npx를 사용할 수 있는 상태이다.

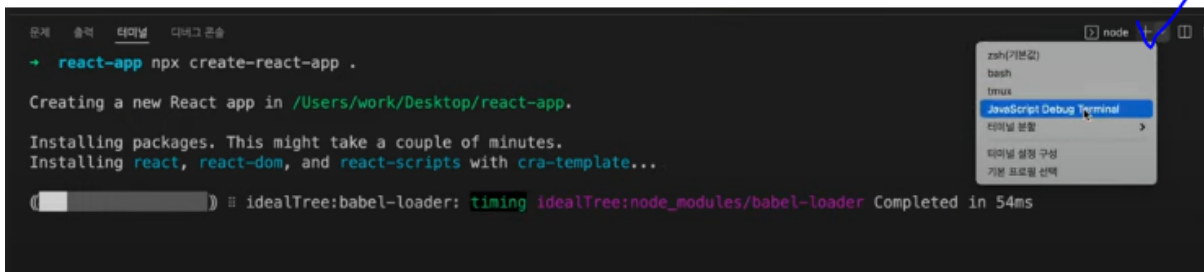
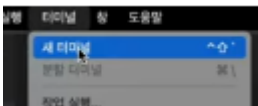
사용중 문제 발생시 관리자 권한으로 실행

visual studio code 설치

<https://code.visualstudio.com/>



바탕화면에 **react**폴더를 만들고 해당 폴더를 **visual studio code**에서 선택한 다음 터미널에 새터미널로 연다.



새터미널에서 다음과 같은 **npx**를 사용한다. 터미널에 문제가 발생하면 왼쪽 상단에서 다른 것을 선택해서 해보자. 파워셸이 제대로 동작하지 않는 경우가 있다. 되도록 관리자 권한으로 실행,리부팅, 캐시정리, **npm**재설치 등을 진행해 보자.

- **npx**로 **create-react-app** 새 프로젝트 생성하기

react폴더안의 ch01폴더를 만들고 터미널로 해당 폴더로 이동한다.

```
cd ch01
```

바탕화면에 폴더를 만들고 이동해서 `npx create-react-app .` 으로 현재폴더에 react 프로젝트를 생성해 보자.

`npx create-react-app .` 현재 폴더에 설치

`npx create-react-app ch01` ch01에 설치

비주얼 스튜디오 코드를 관리자 권한으로 실행했는데도 관리자 권한 문제 에러가 날수 있다.

```
PS C:\Users\park\Desktop\react> npx create-react-app
npx : 이 시스템에서 스크립트를 실행할 수 없으므로 C:\Program Files\nodejs\npx.ps1 파일을 로드할 수 없습니다. 자세한 내용은 about_Execution_Policies(https://go.microsoft.com/fwlink/?LinkID=135170)를 참조하십시오.
위치 줄:1 문자:1
+ npx create-react-app
+ ~~~
+ CategoryInfo          : 보안 오류: (:) [], PSSecurityException
+ FullyQualifiedErrorId : UnauthorizedAccess
PS C:\Users\park\Desktop\react> Get-ExecutionPolicy -List
```

관리자 권한으로 Visual Studio Code를 실행해도 해결되지 않았다면, 다른 방법을 시도해 볼 수 있습니다.

1. PowerShell에서 실행 정책 확인:

먼저 PowerShell에서 현재 실행 정책을 확인하려면 다음 명령어를 입력합니다:

```
Get-ExecutionPolicy -List
```

2. 실행 정책을 변경:

RemoteSigned로 설정되어 있어야 **npx**가 실행됩니다. 실행 정책을 변경하려면 아래 명령어를 실행하세요:

Set-ExecutionPolicy RemoteSigned -Scope CurrentUser

-
- 그런 다음, **Y**를 입력하여 변경을 승인합니다.

3. 정책 변경 후 **Visual Studio Code** 다시 실행:

- 실행 정책을 변경한 후, **Visual Studio Code**를 종료하고, 다시 관리자 권한으로 열어서 터미널에서 **npx create-react-app** 명령어를 실행해 보세요.

버전에 문제가 발생하면 다음과 같은 에러가 날수 있다.

```
PS C:\Users\park\Desktop\react> npx create-react-app ch01

You are running `create-react-app` 5.0.1, which is behind the latest release (5.1.0).

We recommend always using the latest version of create-react-app if possible.

The latest instructions for creating a new app can be found here:
https://create-react-app.dev/docs/getting-started/

npm notice
npm notice New major version of npm available! 9.5.1 -> 11.2.0
npm notice Changelog: https://github.com/npm/cli/releases/tag/v11.2.0
npm notice Run npm install -g npm@11.2.0 to update!
npm notice
PS C:\Users\park\Desktop\react> []
```

node.org에서 **node**프로그램을 최신으로 설치한후 해보자. 설치 불가능시 다음 방법을 사용해 보자.

그래도 해결이 안되면 다음 방법을 참고해 보자.

메시지는 정보 제공과 몇 가지 제안이 포함된 내용입니다:

create-react-app 버전이 오래됨: 현재 사용 중인 **create-react-app** 버전은 **5.0.1**로, 최신 버전인 **5.1.0**보다 뒤쳐져 있습니다. 최신 버전을 사용하는 것이 좋습니다. 글로벌로 **create-react-app**을 업데이트하려면 아래 명령어를 실행하세요:

```
npm install -g create-react-app
```

npm의 새 버전: **npm**의 새로운 주요 버전(**11.2.0**)이 출시되었습니다. **npm**을 업데이트하려면 다음 명령어를 실행하세요:

```
npm install -g npm@11.2.0
```

작업 중 다음과 같은 에러가 날 수 있다.

4 packages are looking for funding
run `npm fund` for details

```
PS C:\Users\park\Desktop\react> npm install -g npm@11.2.0
```

```
npm ERR! code EBADENGINE
```

```
npm ERR! engine Unsupported engine
```

```
npm ERR! engine Not compatible with your version of node/npm: npm@11.2.0
```

```
npm ERR! notsup Not compatible with your version of node/npm: npm@11.2.0
```

```
npm ERR! notsup Required: {"node":"^20.17.0 || >=22.9.0"}
```

```
npm ERR! notsup Actual:   {"npm":"9.5.1","node":"v18.16.1"}
```

```
npm ERR! A complete log of this run can be found in:
```

```
npm ERR!     C:\Users\park\AppData\Local\npm-cache\_logs\2025-03-10T16_35_10_349Z-debug-0.log
```

```
PS C:\Users\park\Desktop\react> []
```

1. Node.js 최신 버전 설치:

- [Node.js 공식 웹사이트](#)에서 최신 LTS 버전(현재는 20.x 이상)을 다운로드하여 설치합니다.

2. npm 업데이트:

Node.js를 최신 버전으로 업데이트한 후, 아래 명령어로 npm을 업데이트합니다:

```
npm install -g npm@11.2.0
```

이렇게 하면 npm 버전 호환성 문제가 해결되고, 최신 버전으로 업데이트할 수 있습니다.

```
→ react-app npx create-react-app .
```

```
Creating a new React app in /Users/work/Desktop/react-app.
```

```
Installing packages. This might take a couple of minutes.
```

```
Installing react, react-dom, and react-scripts with cra-template...
```

```
cd ch01
```

```
npm start
```

설치가 끝나고 `npm start`를 입력하면 우리가 만든 리액 페이지가 3000포트로 접속할 수 있도록 실행된다.

문제 출력 터미널 디버그 콘솔

```
? Something is already running on port 3000. Probably:
```

```
/usr/local/Cellar/node/16.6.2/bin/node /Users/work/dev/project/seomald 79533)
```

```
in /Users/work/dev/project/seomal/static/app
```

```
Would you like to run the app on another port instead? > (Y/n)
```

이미 사용하고 있는 포트가 있으면 상의 이미지 y를 눌러서 새로운 포트번호를 자동발급되거나 입력받을수 있다.

정상적으로 설치되었다면 브라우저에 다음과 같은 이미지를 확인할 수 있다.



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

Compiled successfully!

You can now view ch01 in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.35.106:3000

Note that the development build is not optimized.
To create a production build, use npm run build.

webpack compiled successfully
█
```

상위 처럼 실행 중이 아니라면 localhost:3000 주소 요청시 에러 페이지가 나온다.

컨트롤 +c 를 이용해서 실행중인 서버를 중단 할 수 있다.

요청 에러 발생시 현재 서버가 동작중인지 확인해 보자.

문제 발생시 다음을 확인해 보자.

에러시 확인해볼 사항 관리자 권한인지 확인해 본다.

한글폴더가 있는지 확인해 본다.

리부팅하고 실행해 본다.

npm 캐시 정리

npm 캐시를 정리해서 캐시와 관련된 문제를 해결할 수 있습니다.

```
npm cache clean --force
```

npm 재설치

npm을 재설치해서 문제를 해결할 수 있습니다.

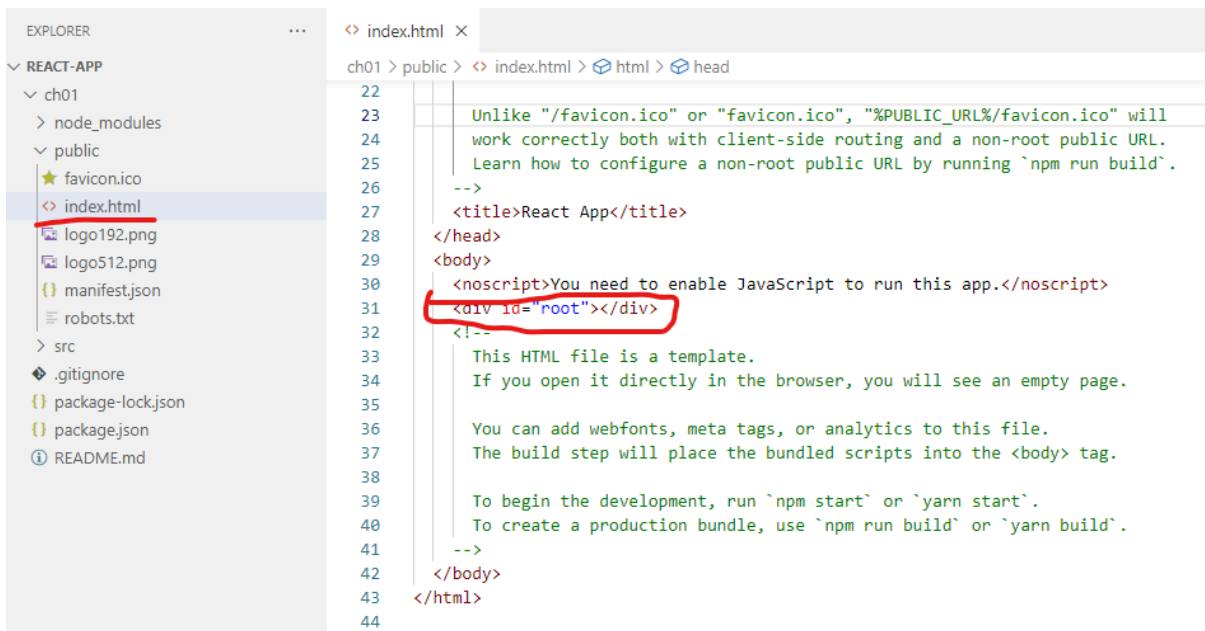
```
npm install -g npm
```

visual studio 의 ch01파일을 열어 다음 파일들을 확인해 보자.

React 프로젝트의 **public** 폴더는 애플리케이션에서 외부 파일을 저장하고 제공하는 용도로 사용됩니다. 이 폴더에 있는 파일들은 빌드 프로세스를 거치지 않고 그대로 배포되며, 클라이언트에서 직접 접근할 수 있습니다. 주요 용도는 다음과 같습니다:

- 정적 파일 저장: 이미지, 폰트, 아이콘 등 **React** 컴포넌트와 별도로 관리되는 파일을 저장합니다.
- 직접 접근 가능: URL을 통해 외부에서 직접 접근할 수 있습니다

index.html이 실제 시작 실행 파일이다. index.html에 <div id="root"></div>가 다른 파일(index.js)에서 렌더링된 html이 들어 가는 부분이다. 핵심 파일 이지만 사용할 일이 없는 파일이다.



index.html: React 애플리케이션의 기본 HTML 템플릿, **root div**를 포함.

index.js: React 앱을 브라우저에 렌더링하는 엔트리 포인트.

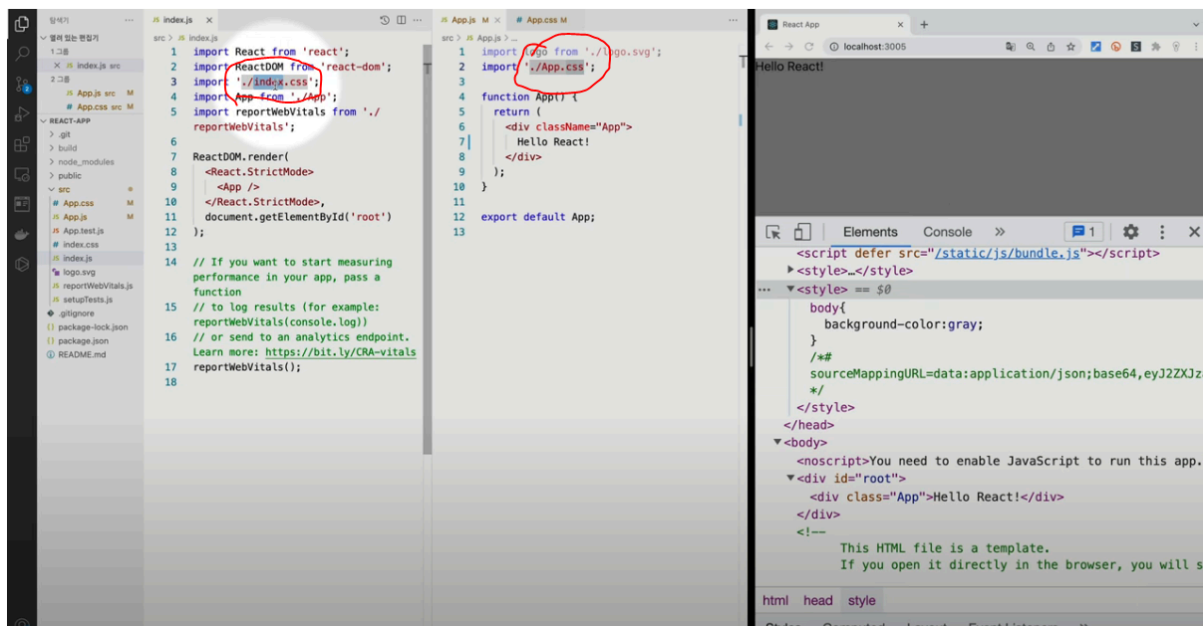
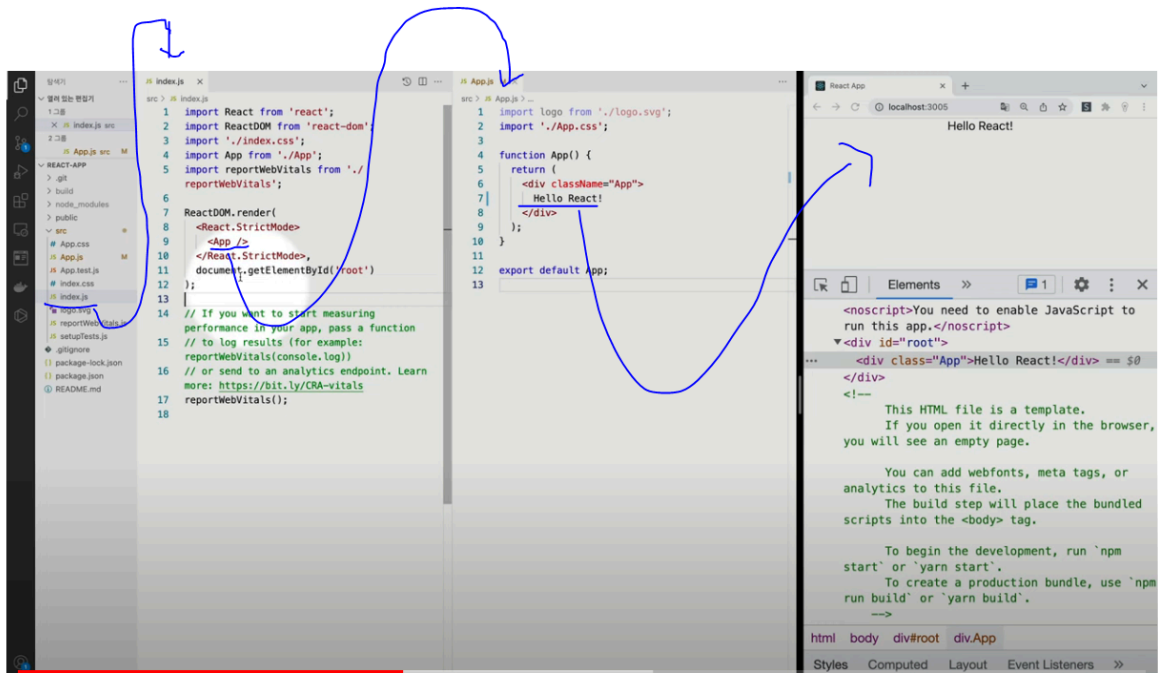
App.js: 애플리케이션의 주요 컴포넌트로, UI와 로직을 정의.

App.js를 다음과 같이 변경해 보자.

```
import logo from './logo.svg';
import './App.css';
function App() {
  return (
    <div className="App">

      Hello React?

    </div>
  );
}
export default App;
```



css를 사용하고 싶다면 해당 css안에서 작업해야 한다.

`index.css`와 `app.css`는 React 프로젝트에서 스타일을 관리하는 데 사용

`index.css`

전체 애플리케이션에 적용되는 전역 스타일을 정의하는 데 사용됩니다.
글로벌하게 적용되므로, 어느 컴포넌트에서든지 이 스타일이 적용됩니다.

`app.css`

App 컴포넌트에 특정한 스타일을 정의하는 데 사용됩니다.

코드 정리방법

ctrl+ A 모두 선택

alt +shift + f 코드정리

index.css에 기술하면 app.js에서 div 색상이 변경된다. 모든 html 파일의 태그에 css가 적용된다.

```
div{  
  background-color: blue;  
}
```

app.css에 기술하면 app.js에서 만 적용 된다.

```
div{  
  background-color: red;  
}
```

모듈 시스템 3가지 핵심 문법 정리

1. Default Export 2. Named Export 3. Import

하나의 파일을 분리해서 여러개의 파일로 기술하는 방식을 의미한다.

- 1. **Default Export** 임포트한 파일에서 대표기능 한개만 가져다가 쓸때 사용
- 2. **Named Export** 임포트한 파일에서 여러개의 기능을 가져다가 쓸때 사용
- 3. **Import** 임포트한 파일을 전체 적용해 가져다가 쓸때 사용

```
import App from './App';  
  
import { render, screen } from '@testing-library/react';  
  
import logo from './logo.svg'; //뒤에 있는 이미지를 식별자 logo로 사용한다.  
  
import './App.css';
```

핵심 비교표

구분	주요 특징	사용 예시	가져오기 방식
----	-------	-------	---------

Default	모듈의 주요 기능, 이름 자유 변경	<code>export default</code> 컴포넌트	<code>import 별명 from</code> '경로'
Named	다중 항목 내보내기, 이름 고정	<code>export { 유틸1,</code> <code>유틸2 }</code>	<code>import { 이름 } from</code> '경로'
CSS	스타일시트 적용	여러 css 기술	<code>import './App.css'</code>

✓ 1. Default Export

문법 설명: 모듈에서 하나의 기본 값만 `export default`로 내보냄
// 내보내기

```
const 이름1 = 값;  
export default 이름1;
```

```
export default function 이름2() {}
```

```
function 이름3() {}  
export default 이름3
```

```
// 가져오기  
import 사용자지정이름(이름1) from '모듈경로';
```

✓ 2. Named Export

문법 설명: 여러 값을 이름을 붙여 `export`로 내보냄
// 내보내기

```
export const 이름 = 값;
```

```
export function 이름() {}
```

```
const 항목1 = 값;
```

```
function 항목2() {}
```

```
export { 항목1, 항목2 }; //선언후 한번에 export할 수 있다.
```

```
// 가져오기
```

```
import { 이름1, 이름2 } from '모듈경로';
```

✅ 3. CSS Import

문법 설명: JavaScript나 React 파일에서 CSS 파일을 불러와 스타일을 적용 js 파일이면 해당 파일의 내용을 그대로 가져온다

```
// CSS 파일 가져오기
```

```
import '파일경로/파일이름.css';
```

```
import '파일경로/파일이름.js';
```

Default/Named Export 확장 설명 (CSS import 포함)

1. Default/Named Export 확장 예제

1.1 Default Export를 함수 선언과 함께 바로 사용

```
// 📁 utils.js
```

```
// Default export (함수 선언과 동시에 내보내기)
```

```
export default function mainUtil() {
```

```
    return "MAIN FUNCTION";
```

```
}
```

```
// Named exports
```

```
export function helper1() {
```

```
    return "HELPER 1";
```

```
}

export function helper2() {

    return "HELPER 2";

}
```

1.2 Import 사용 예제

```
// 📁 app.js

import primaryFunction, { helper1, helper2 as h2 } from './utils.js';

console.log(primaryFunction()); // "MAIN FUNCTION"

console.log(helper1());          // "HELPER 1"

console.log(h2());               // "HELPER 2" (별칭 사용)

import logo from './logo.svg';

import './App.css';

import mainUtil from './utils';

import MyUtil from './utils';

//import { helper1, helper2 } from './utils';

import { helper1, helper2 as h2 } from './utils';

function App() {

    // alert(mainUtil());

    // alert(MyUtil());

    alert(helper1());

    //Calert(helper2());

    alert(h2());

}
```

```
    return <div className="App">Hello world!</div>;  
  }  
  
export default App;
```

2. CSS 파일 import 방법

2.1 기본 CSS import

```
// 📁 component.js  
  
import './styles.css'; // CSS 파일을 통으로 import (웹팩 등 모듈 번들러 필요)  
  
import './styles.js'; // js 파일을 통으로 import (웹팩 등 모듈 번들러 필요)
```

2.2 CSS Module 사용 (React 예제)

```
import styles from './Button.module.css'; // CSS Module import  
  
export default function Button() {  
  
  return (  
  
    <button className={styles.primary}>  
  
      Click Me  
  
    </button>  
  
  );  
}
```

4. 핵심 비교표 (업데이트 버전)

특징	Default Export	Named Export	CSS Import
문법	<code>export default function</code>	<code>export function</code>	<code>import './file.css'</code>
import 방식	<code>import XXX from...</code>	<code>import { XXX } from...</code>	<code>import styles from...</code> (CSS Modules)
이름 변경	자유롭게 변경 가능	<code>as</code> 로 별칭 지정 필요	CSS 클래스명 그대로 사용
사용 예시	주 컴포넌트/메인 기능	유틸 함수, 상수	스타일 적용
파일당 개수	1개	여러 개	여러 개 import 가능

번외 설명: 다양한 Import/Export 패턴

1. 모듈 전체 import

```
import * as mathUtils from './utils/mathUtils';
```

```
console.log(mathUtils.add(2, 3));
```

2. 이름 변경하여 import

```
import { add as sum } from './utils/mathUtils';

console.log(sum(2, 3));
```

3. CSS 모듈 (CSS-in-JS)

// 웹팩 *css-loader* 설정이 되어있다면

```
import styles from './styles/main.module.css';

const element = document.createElement('div');

element.className = styles['main-container']; // 고유한 클래스명이 생성됨
```

4. 동적 import (코드 스플리팅)

// 필요할 때만 모듈 로드

```
button.addEventListener('click', async () => {

  const math = await import('./utils/mathUtils');

  console.log(math.multiply(4, 5));

});
```

5. .js 생략

```
import App from './App';
```

> 03. JSX의 기본구조

JSX란? REACT에서 컴포넌트 만들때 사용하는 언어

REACT에서 컴포넌트란? 복잡한 HTML 코드를 하나의 HTML태그로 만든 것

React JSX의 기초 문법을 설명해드리겠습니다.

JSX는 JavaScript 코드 내에서 HTML과 유사한 문법을 사용하여 React 컴포넌트를 정의하고 사용할 수 있게 해줍니다.

"React 컴포넌트(JSX 사용)는 .jsx, 순수 JavaScript는 .js로 TypeScript 컴포넌트 .tsx 확장자 순수 TypeScript는 .ts를 사용

1. jsx 기본구조

1. React은 컴포넌트로 구성되어 있다.
2. 컴포넌트는 하나의 자바스크립트 파일에 하나의 함수로 만든다.
3. 하단에 **export default** 함수이름;을 넣는다.
4. 컴포넌트는 화면 구성을 위한 데이터 조작 부분을 메소드안에 기술한다.
5. 메소드안에 기술된 자바스크립트 코드를 가지고 리턴 값에 화면을 표현한다.
6. 컴포넌트 이름은 반드시 대문자로 시작해야 한다.

1. 컴포넌트 중심 구조

- "React는 컴포넌트(UI 조각)로 구성된다"
 - 모든 화면은 독립된 컴포넌트의 조합으로 생성됩니다.
 - 예: Header, Button, Card 등

2. 파일 구성 규칙

- "1파일 = 1컴포넌트 = 1함수"

// 📁 Button.jsx

```
function Button() { // 컴포넌트 함수
  return <button>Click</button>;
}
```

- **export default Button;** // 필수 내보내기

3. 컴포넌트 작동 원리

- "로직(JS) + 화면(JSX)을 함께 작성"

```
function Counter() {
  // 1. 데이터 조작 (JavaScript)
  const count=0;

  // 2. 화면 반환 (JSX)
  return (
    <div>
      <p>{count}</p>
    </div>
  );
}
```

컴포넌트

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

function App() {
  const element = <h1>Hello, world!</h1>;

  return (
    <div className="App">
      {element}
    </div>
  );
}

export default App;
```

구조

화면

4. 컴포넌트 네이밍 규칙

- "반드시 대문자로 시작 (PascalCase)"
 - Button (○) / button (✗)
 - 소문자로 시작하면 HTML 태그로 인식됩니다.

5. 필수 포함 요소

요소	설명	예시
함수 선언	컴포넌트 로직과 JSX 포함	function Component() {}
export default	외부에서 사용 가능하도록 내보냄	export default Component;
JSX 반환	화면을 정의하는 XML-like 구문	return <div>Content</div>;

2. JSX 표현식

JSX 내부에서는 종괄호 `}`를 사용하여 JavaScript 표현식을 삽입할 수 있습니다.

```
const name = 'John';
const element = <h1>Hello, {name}!</h1>;
```

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

function App() {
  const name = 'John';
  const element = <h1>Hello, {name}!</h1>;

  return (
    <div className="App">
      {element}
    </div>
  );
}

export default App;
```

3. JSX 요소 닫기

모든 JSX 태그는 반드시 닫혀야 합니다. 단일 태그는 **self-closing** 태그로 작성해야 합니다.

// 잘못된 예시

```
// const element = ;
```

// 올바른 예시

```
const element = ;
```

4. JSX 속성

JSX에서 속성은 HTML과 비슷하게 작성되지만, 일부 속성 이름은 JavaScript의 관습에 따라 작성됩니다. 예를 들어, **class** 대신 **className**, **for** 대신 **htmlFor**를 사용합니다.

javascript 키워드와 겹치면 다른 식별자를 사용한다.

```
const element = <div className="container"></div>;
// ✅ 'class' 대신 'className'
const label = <label htmlFor="username">Username</label>;
// ✅ 'for' 대신 'htmlFor'
```

5. 주석

JSX 내에서 주석: JSX 내에서 주석은 중괄호 안에 **/**/**를 이용해서 주석을 만든다.

한줄주석 // 은 제공되지 않는다.

```
const element = ( <div> { /* This is a comment in JSX */ } <p>Hello, World!</p> </div> );
```

6. 루트는 반드시 하나여야 한다.

두개 이상 리턴해야 할 경우 `<></>`안에 기술한다.

Fragments: JSX에서 두 개 이상의 요소를 반환하려면 반드시 하나의 부모 요소로 감싸야 합니다. 그러나 때로는 부모 요소를 추가하지 않고 여러 요소를 그룹화하고 싶을 때가 있습니다. **React Fragment**를 사용하여 부모 요소 없이 여러 요소를 그룹화할 수 있습니다.

```
const element = ( <> <p>Paragraph 1</p> <p>Paragraph 2</p> </> );
```

```
import logo from './logo.svg';
import './App.css';
function App() {
  const name = 'John';
  const element = <h1>Hello, {name}!</h1>;
  return (
    <>
      <div className="App">
        {element}
      </div>
      <div className="App">
        {element}
      </div>
    </>
  );
}
export default App;
```

7. 삼항연산자 사용 가능

조건부 렌더링: **JSX** 내에서 조건부로 요소를 렌더링하려면 삼항 연산자나 논리 연산자를 사용할 수 있습니다.

```
jsx
const isLoggedIn = true;
const element = isLoggedIn ? <p>Welcome, User!</p> : <p>Please log in</p>;
```

8. **JSX**로 컴포넌트 만들어 사용하기

컴포넌트는 단순히 함수를 정의하고 **JSX**를 반환해서 만들 수 있다.

```
function Greeting() {
  return <h1>Hello, world</h1>;
}
```

함수로 컴포넌트를 만들어 원할때 사용할 수 있다.

```

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

function Greeting() {
  return <h1>Hello, world</h1>;
}

function App() {
  const name = 'John';
  const element = <h1>Hello2, {name}!</h1>;
  return (
    <div className="App">
      {element}
      <Greeting/>
      <Greeting/>
    </div>
  );
}

export default App;

```

9. Props

컴포넌트에 데이터를 전달하기 위해 **props**를 사용합니다. **props**는 컴포넌트의 속성으로 전달됩니다.

컴포넌트 속성으로 선언한 속성이름으로 속성의 값을 읽어 올수 있다. **props.속성이름** 하면 속성의 값에 접근한다.

```

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

```

// 사용 예시

```
<Greeting name="Alice" />
```

```

import React from 'react';
import './App.css';

```

// Greeting 컴포넌트 정의

```

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

```

// App 컴포넌트 정의

```

function App() {
  return (
    <div className="App">
      <Greeting name="Alice" />
      <Greeting name="Tom" />
    </div>
  );
}

```

```
);  
}
```

```
export default App;
```

문제 1)

`<GreetingProps name= "Tom" age=15/>`일때 “Tom의 나이는 15입니다.” 가 출력 되도록 컴포넌트를 만들어 보자.

10. 자식 요소

컴포넌트에 자식 요소를 포함할 수 있습니다.

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

// 사용 예시

```
<Welcome name="Alice">  
  <p>This is a child element.</p>  
</Welcome>
```

11. 조건부 렌더링

JSX 내에서 조건부로 요소를 렌더링할 수 있습니다.

```
function Greeting(props) {  
  if (props.isLoggedIn) {  
    return <h1>Welcome back!</h1>;  
  } else {  
    return <h1>Please sign up.</h1>;  
  }  
}
```

```
import React from 'react';  
import './App.css';
```

// Greeting 컴포넌트 정의

```
function Greeting(props) {  
  if (props.isLoggedIn) {  
    return <h1>Welcome back!</h1>;  
  } else {  
    return <h1>Please sign up.</h1>;  
  }  
}
```

// App 컴포넌트 정의

```
function App() {  
  const userIsLoggedIn = true; // 로그인 상태 변수(true/false로 변경 가능)  
  return (  
    <div className="App">  
      <Greeting isLoggedIn={userIsLoggedIn} />  
    </div>  
  );  
}  
export default App;
```

> 04. 화살표 함수 사용법

화살표 함수는 **JavaScript**에서 함수를 간단하게 만드는 방법입니다. 기존의 함수를 좀 더 짧고 간결하게 작성할 수 있게 도와줍니다. 두 방식의 결과의 차이점은 없시 모양만 다르다.

```
function 함수이름(매개변수1, 매개변수2, ...) {  
    // 함수 본문  
    // return 리턴할값  
}
```

```
const 함수이름 = (매개변수1, 매개변수2, ...) => {  
    // 함수 본문  
    // return 리턴할값  
};
```

```
// 두 예제 비교
```

```
// 일반 함수
```

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

```
alert(add());
```

```
// 화살표 함수
```

```
const add = (a, b) => {  
    return a + b;  
}
```

```
alert(add());
```


1. 기본 형태

화살표 함수:

```
const greet = (name) => {  
    console.log(`Hello, ${name}!`);  
};  
  
greet('Alice'); // 출력: Hello, Alice!
```

일반 함수:

```
function greet(name) {  
    console.log(`Hello, ${name}!`);  
}  
  
greet('Alice'); // 출력: Hello, Alice!
```

다양한 형태의 화살표 함수

// 1. 매개변수 없을 때

```
const 함수명 = () => { 실행문 };
```

// 2. 매개변수 1개일 때 (괄호 생략 가능)

```
const 함수명 = 매개변수 => { 실행문 };
```

// 3. 매개변수 여러 개일 때

```
const 함수명 = (매개변수1, 매개변수2) => { 실행문 };
```

// 4. 리턴값만 있을 때 (중괄호 & return 생략 가능)

```
const 함수명 = (a, b) => a + b;
```

```
const 함수명 = (a, b) => { return a + b};
```

2. 매개변수가 없을 때

일반 함수:

```
function sayHello() {  
    console.log('Hello, world!');  
}  
  
sayHello(); // 출력: Hello, world!
```

화살표 함수:

```
const sayHello = () => {  
    console.log('Hello, world!');  
}  
  
sayHello(); // 출력: Hello, world!
```

3. 매개변수가 하나일 때

일반 함수:

```
function square(x) {  
    return x * x;  
}  
  
console.log(square(5)); // 출력: 25
```

화살표 함수:

```
const square = x =>{
```

```
        return x * x;
    }
    console.log(square(5)); // 출력: 25
```

4. 함수 본문이 하나의 표현식일 때

중괄호와 **return**를 생략할 수 있다.

화살표 함수:

```
const add = (a, b) => a + b;
console.log(add(3, 4)); // 출력: 7
```

일반 함수:

```
function add(a, b) {
    return a + b;
}
console.log(add(3, 4)); // 출력: 7
```

5. 함수 본문이 여러 줄일 때

일반 함수:

```
function multiply(a, b) {
    const result = a * b;
    return result;
}
console.log(multiply(6, 7)); // 출력: 42
```

화살표 함수:

```
const multiply = (a, b) => {  
    const result = a * b;  
    return result;  
};  
  
console.log(multiply(6, 7)); // 출력: 42
```

3. 리턴 생략

일반 함수:

```
function square(x) {  
    return x * x;  
}  
  
console.log(square(5)); // 출력: 25
```

화살표 함수:

```
const square = x =>{  
    return x * x;  
}  
  
console.log(square(5)); // 출력: 25  
  
const square = x => x * x;
```

기존 함수:

```
function greet(name) {  
    return "안녕하세요, " + name + "님!";  
}
```

화살표 함수:

```
var greet = name => "안녕하세요, " + name + "님!";
```

다음은 화살표 함수 설명하는 내용입니다.

함수 선언: **function** 키워드 대신 **=>** 기호를 사용하여 함수를 선언합니다.

인자(매개변수): 화살표 앞쪽에 인자가 들어 간다. 인자가 없으면(), 여러 개의 인자가 있을 때는 괄호로 감싸줍니다. 예를 들어 (x, y)=>, 하나이면 중괄호를 생략할 수 있다. 예를 들어, x =>

함수 본문: 함수 본문은 화살표(=>) 다음에 나옵니다. 본문이 한줄일 경우 중괄호 {}로 감싸지 않아도 됩니다.

리턴 값: 본문이 한줄일 경우 별도의 **return** 키워드를 사용하지 않아도 됩니다.

간단한 예제로 화살표 함수를 설명하겠습니다:

```
// 기존 함수
```

```
function add(x, y) {  
    return x + y;  
}
```

```
// 화살표 함수로 변환
```

```
var add = (x, y) => x + y;
```

```
//(x,y)=>{x=x+y;return x;}; //같은 의미
```

위 예제에서, add 함수를 화살표 함수로 변환하면 더 간결한 코드가 됩니다. 화살표 함수는 주로 익명 함수로 사용되며, 함수를 변수에 할당하거나 배열의 **map**, **filter**, **reduce**와 같은 고차 함수와 함께 사용되어 코드를 더 읽기 쉽게 만들어줍니다. */

```
var result=arr1.filter(x=>x%2===1);
```

자바스크립트 화살표 함수(Arrow Function) 예제 10개를 준비했습니다. 초급부터 중급 수준까지 다양하게 넣었어요:

- ◆ 1. 두 수 더하기

javascript

복사편집

```
const add = (a, b) => a + b;  
console.log(add(2, 3)); // 5
```

◆ 2. 인사말 반환

javascript

복사편집

```
const greet = name => `Hello, ${name}!`;   
console.log(greet("철수")); // Hello, 철수!
```

◆ 7. 문자열 길이 반환

javascript

복사편집

```
const getLength = str => str.length;  
console.log(getLength("화살표 함수")); // 6
```

◆ 8. setTimeout 사용

javascript

복사편집

```
setTimeout(() => {  
  console.log("3초 뒤 출력");  
}, 3000);
```

◆ 9. 삼항 연산자와 함께 사용

```
const isAdult = age => age >= 18 ? "성인" : "미성년자";  
console.log(isAdult(20)); // 성인
```

> 04. css 사용

13. 인라인 스타일링

react에서 여러가지 css사용하기

1. Inline Styles

인라인 스타일을 사용하면 **JavaScript** 객체로 스타일을 직접 지정할 수 있습니다.

JSX에서 스타일을 적용할 때는 객체를 사용합니다.

```
const divStyle = {  
  color: 'blue',  
  backgroundColor: 'lightgrey'  
};  
const element = <div style={divStyle}>Styled element</div>;
```

```
import React from 'react';  
import './App.css';
```

// 스타일 객체 정의

```
const divStyle = {  
  color: 'blue',  
  backgroundColor: 'lightgrey'  
};
```

// App 컴포넌트 정의

```
function App() {  
  return (  
    <div className="App">  
      <div style={divStyle}>Styled element</div>  
    </div>  
  );  
}  
export default App;
```

다음 코드를 읽고 이해해 보자.

```
import './App.css';
function App() {
  const name = "Tom";
  const naver = {
    name: "네이버",
    url: "https://naver.com",
  };
  return (
    <div className="App">
      <h1
        style={{
          color: "#f0f",
          backgroundColor: "green",
        }}
      >
        Hello, {name}.<p>{2 + 3}</p>
      </h1>
      <a href={naver.url}>{naver.name}</a>
    </div>
  );
}
export default App;
```

2. 일반 CSS 파일 사용

가장 기본적인 방법으로, CSS 파일을 작성하고 컴포넌트에서 이를 임포트하여 사용하는 방식입니다.

App의 하위 컨트롤러에도 영향을 미친다.

```
/* styles.css */
.container {
  background-color: lightblue;
  padding: 20px;
}
```

// App.js


```
import React from 'react';
import './styles.css';
function App() {
  return <div className="container">Hello, World!</div>;
}
```

```
export default App;
```

현 상태에서 App에서 사용하는 하위 App2 컴포넌트에 .container를 사용하는 DIV를 추가하면 APP div와 하위에 있는 App2 div 둘다 .container의 영향을 받는다.

// App.js 를 변경하고 실행해 보자.

```
import React from 'react';
import './styles.css';
function App2() {
  return <div className="container">Hello, World!</div>;
}
function App() {
  return <div>
    <div className="container">Hello, World!</div>
    <App2/>
  </div>;
}
export default App;
```

3. CSS Modules

.module.css는 특정한 기능을 수행하기 위해 약속된 확장자입니다. React에서 CSS Modules을 사용할 때, .module.css 확장자를 가진 파일을 사용합니다.

```
css
/* App.module.css */
.container {
  background-color: lightblue;
  padding: 20px;
}
/*A{}
#hello
  module.css에서는 사용하지 않을것을 권장
*/
// App.js
import React from 'react';
import styles from './App.module.css';

function App() {
  return <div className={styles.container}>Hello, World!</div>;
```

```
}
```

```
export default App;
```

현 상태에서 App에서 사용하는 하위 App2 컴포넌트에 .container를 사용하는 DIV를 추가하면 App.module.css 에 정의한 내용은 App에 있는 div에만 영향을 주고 하위에 있는 App2 div에는 영향을 주지 않는다.

```
// App.js 변경
```

```
import React from 'react';
```

```
import styles from './App.module.css';
```

```
function App2() {
```

```
  return <div className="container">Hello, App2!</div>;
```

```
}
```

```
function App() {
```

```
  return <div>
```

```
    <div className={styles.container}>Hello, App!</div>
```

```
    <App2/>
```

```
  </div>;
```

```
}
```

```
export default App;
```

> 05. 변수와 scope

1. 변수 선언 키워드

var let const

1) var (ES5 및 이전 버전)

- 함수 범위(Function Scope)를 가짐 (if, for 밖에서 접근가능)
- 재선언(같은 이름의 변수 선언) 가능
- 선언 전에 사용할 수 있음 (호이스팅 시 undefined 할당)
- var는 최신 JavaScript(ES6+)에서는 잘 사용되지 않음

javascript

복사편집

```
console.log(a); // undefined (호이스팅 발생)
```

```
var a = 10;
```

```
console.log(a); // 10
```

```
var b = 20;
```

```
var b = 30; // 재선언 가능
```

```
console.log(b); // 30
```

2) let (ES6 도입)

- 블록 범위(Block Scope)를 가짐 ({ } 내부에서만 유효)
- 재선언 불가능 (같은 이름의 변수 중복 선언 불가)
- 선언 전에 사용할 수 없음 (TDZ - Temporal Dead Zone 발생)

javascript

복사편집

```
console.log(x); // ReferenceError (TDZ로 인해 접근 불가)
```

```
let x = 10;
```

```
console.log(x); // 10
```

```
let y = 20;
```

```
let y = 30; // SyntaxError: Identifier 'y' has already been declared
```

3) `const` (ES6 도입)

- 블록 범위(Block Scope)를 가짐 (`let`과 동일)
- 반드시 선언과 동시에 초기화해야 함
- 재할당 불가능 (변경 불가능한 상수 변수)

javascript

복사편집

```
const z = 100;
```

```
z = 200; // TypeError: Assignment to constant variable.
```

```
const pi; // SyntaxError: Missing initializer in const declaration
```

✚ 단, `const`는 객체(Object)와 배열(Array)일 때 내부 값은 변경할 수 있음

javascript

복사편집

```
const person = { name: "Alice" };
```

```
person.name = "Bob"; // 가능
```

```
console.log(person); // { name: "Bob" }
```

함수에 대한 설명

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

JavaScript 변수 선언 키워드와 함수 범위(Function Scope) 관련 내용

JavaScript에서 변수를 선언하는 방법(**var**, **let**, **const**)은 **함수 범위(Function Scope)와 블록 범위(Block Scope)**에 영향을 미칩니다. 여기서는 함수 범위와 관련된 내용을 추가하여 설명하겠습니다.

1. 함수 범위(Function Scope)란?

함수 범위(Function Scope)는 변수가 선언된 함수 내부에서만 접근할 수 있는 범위를 의미합니다. 즉, 함수 내부에서 선언된 변수는 함수 외부에서 접근할 수 없습니다.

javascript

복사편집

```
function myFunction() {  
    var a = 10; // 함수 내부에서 선언된 변수  
    let b = 20;  
    const c = 30;  
    console.log(a, b, c); // 10, 20, 30  
}
```

```
myFunction();
```

```
console.log(a); // ReferenceError: a is not defined
```

```
console.log(b); // ReferenceError: b is not defined
```

```
console.log(c); // ReferenceError: c is not defined
```

✅ 결론: 함수 내부에서 선언된 변수는 함수가 종료되면 사라지며, 함수 외부에서는 접근할 수 없음.

2. var는 "함수 범위"를 가짐

var로 선언된 변수는 ****함수 범위(Function Scope)****를 가집니다.

즉, if, for 등의 블록 {} 내부에서 선언해도 함수 전체에서 접근할 수 있습니다.

javascript

복사편집

```
function example() {  
    if (true) {  
        var testVar = "I am inside if block";  
    }  
    console.log(testVar); // "I am inside if block" (함수 전체에서 접근 가능)  
}
```

```
example();
```

```
console.log(testVar); // ReferenceError: testVar is not defined (함수 외부에서는 접근 불가능)
```

🚨 문제점: if 블록 안에서 선언했지만, 함수 전체에서 접근할 수 있음 → 의도치 않은 변경 위험

3. let과 const는 "블록 범위(Block Scope)"를 가짐

let과 const는 ****블록 범위(Block Scope)****를 가지므로 {} 내부에서만 유효합니다.

javascript

복사편집

```
function example() {
```

```

if (true) {
    let testLet = "I am inside if block";
    const testConst = "I am also inside if block";
}

console.log(testLet); // ReferenceError: testLet is not defined
console.log(testConst); // ReferenceError: testConst is not defined
}

```

example();

✅ 결론: **let**과 **const**는 블록 **{ }** 내부에서만 접근 가능하며, **var**보다 안전함.

4. 함수 내부에서 var, let, const 차이점 정리

선언 키워드	함수 범위(Function Scope)	블록 범위(Block Scope)	중복 선언	호이스팅 시 초기값
var	✅ 있음	❌ 없음	✅ 가능	undefined
let	❌ 없음	✅ 있음	❌ 불가능	TDZ (오류 발생)
const	❌ 없음	✅ 있음	❌ 불가능	TDZ (오류 발생)

✅ **let**과 **const**를 사용하면 함수 내에서 블록 범위를 유지할 수 있어, 의도치 않은 변수 변경을 방지할 수 있음.

🚨 **var**는 함수 전체에서 접근 가능하므로, 되도록 사용하지 않는 것이 좋음.

5. 함수 내부에서 선언하지 않은 변수 (자동 전역 변수)

함수 내부에서 **var**, **let**, **const** 없이 변수를 선언하면 **자동으로 전역 변수(Global Variable)**가 됩니다.

이는 버그의 원인이 될 수 있으므로 사용하지 않는 것이 좋습니다.

javascript

복사편집

```
function myFunction() {  
    globalVar = "I am global!"; // var, let, const 없이 선언 (자동 전역 변수)  
}
```

```
myFunction();
```

```
console.log(globalVar); // "I am global!" (전역 변수가 되어버림)
```

✅ 해결 방법: **let**, **const**를 사용하여 명시적으로 선언해야 함.

```
import React from 'react';
```

이 구문은 **React** 컴포넌트를 정의하고 **JSX**를 사용할 수 있도록 하기 위해 추가됩니다.
최신 버전에서는 추가하지 않아도 **jsx**를 사용할 수 있다.

> 04.배열

하나의 변수에 여러개의 데이터를 넣고 인덱스를 이용해서 여러개의 데이터에 접근하는 방법

1. 배열 선언 및 초기화

배열을 생성하는 방법

// 방법 1: 대괄호([]) 사용 (가장 일반적)

```
let arr1 = [1, 2, 3, 4, 5];
```

// 방법 2: Array() 생성자 사용

```
let arr2 = new Array(1, 2, 3, 4, 5);
```

// 방법 3: 빈 배열 생성 후 값 추가

```
let arr3 = [];
```

```
arr3.push(1);
```

```
arr3.push(2);
```

✓ []을 사용하는 것이 가장 일반적이며, `new Array()`는 권장되지 않습니다.

2. 배열 요소 접근 및 변경

배열 요소는 0부터 시작하는 인덱스(index)를 사용해 접근합니다.

javascript

코드 복사

```
let fruits = ["Apple", "Banana", "Cherry"];
```

// 요소 접근

```
console.log(fruits[0]); // Apple
```

```
console.log(fruits[1]); // Banana
```

```
// 요소 변경
fruits[1] = "Blueberry";
console.log(fruits); // ["Apple", "Blueberry", "Cherry"]
```

✓ 배열의 인덱스는 0부터 시작합니다.

🔥 3. 배열의 주요 속성과 메서드

- ◆ 배열 길이 (`length`)

```
let numbers = [10, 20, 30, 40];
console.log(numbers.length); // 4
```

- ◆ 배열 끝에 요소 추가 (`push`) / 삭제 (`pop`)

javascript

코드 복사

```
let arr = [1, 2, 3];

arr.push(4); // 마지막에 추가
console.log(arr); // [1, 2, 3, 4]

arr.pop(); // 마지막 요소 제거
console.log(arr); // [1, 2, 3]
```

- ◆ 배열 앞에 요소 추가 (`unshift`) / 삭제 (`shift`)

```
let arr = [2, 3, 4];

arr.unshift(1); // 맨 앞에 추가
console.log(arr); // [1, 2, 3, 4]

arr.shift(); // 맨 앞 요소 제거
console.log(arr); // [2, 3, 4]
```

- ◆ 특정 위치에 요소 추가/제거 (**splice**)

```
let arr = ["a", "b", "c", "d"];
```

```
// 1번 인덱스부터 2개 제거
```

```
arr.splice(1, 2);
```

```
console.log(arr); // ["a", "d"]
```

```
// 1번 인덱스에 "x", "y" 추가
```

```
arr.splice(1, 0, "x", "y");
```

`arr.splice(1, 0, "x", "y")`의 의미:

- 1: 시작 인덱스 (1번 인덱스 위치에 삽입할 것)
- 0: 삭제할 요소 개수 (삭제는 하지 않음)
- "x", "y": 삽입할 요소들

```
console.log(arr); // ["a", "x", "y", "d"]
```

📌 4. 배열 순회 (반복문)

- ◆ for 반복문

javascript

코드 복사

```
let arr = ["Apple", "Banana", "Cherry"];
```

```
for (let i = 0; i < arr.length; i++) {  
  console.log(arr[i]);  
}
```

- ◆ `forEach()` 메서드

```
arr.forEach((item, index) => {  
  console.log(index, item);  
});
```

- ◆ `for...of` 반복문 (ES6)

```
const arr = ['apple', 'banana', 'cherry'];
```

```
for (let fruit of arr) {  
  console.log(fruit);  
}
```

```
apple  
banana  
cherry
```

- ◆ 3. 배열의 제공값 만들기 (`map` 사용)

기존배열의 데이터를 가지고 새로운 배열을 만든다.

```
const numbers = [1, 2, 3, 4];  
const squares = numbers.map(num => num * num);  
console.log(squares); // [1, 4, 9, 16]
```

- ◆ 4. 짝수만 필터링

기존 배열에서 원하는 데이터만 뽑아서 새로운 배열을 만들때 사용

```
const nums = [1, 2, 3, 4, 5, 6];  
const evens = nums.filter(n => n % 2 === 0);  
console.log(evens); // [2, 4, 6]
```

◆ 5. 객체 반환 (소괄호로 감싸기)

```
const createUser = (name, age) => ({ name, age });  
console.log(createUser("영희", 25)); // { name: '영희', age: 25 }
```

◆ 6. 배열의 총합 계산 (reduce 사용)

javascript

복사편집

```
const arr = [10, 20, 30];  
const sum = arr.reduce((acc, cur) => acc + cur, 0);  
console.log(sum); // 60
```

find() 메소드

find() 메소드는 배열에서 조건을 만족하는 첫 번째 요소를 반환합니다.

javascript

```
const numbers = [5, 12, 8, 130, 44];  
const found = numbers.find(element => element > 10);
```

```
console.log(found); // 12 (10보다 큰 첫 번째 요소)
```

some() 메소드

some() 메소드는 배열의 요소 중 하나라도 조건을 만족하면 true를 반환합니다.

javascript

```
const numbers = [1, 2, 3, 4, 5];  
const hasEven = numbers.some(element => element % 2 === 0);
```

```
console.log(hasEven); // true (짝수가 적어도 하나 존재함)
```

every() 메소드

every() 메소드는 배열의 모든 요소가 조건을 만족하면 true를 반환합니다.

javascript

```
const numbers = [2, 4, 6, 8, 10];  
const allEven = numbers.every(element => element % 2 === 0);
```

```
console.log(allEven); // true (모든 요소가 짝수)
```

includes() 메소드

includes() 메소드는 배열에 특정 요소가 포함되어 있는지 확인합니다.

javascript

```
const fruits = ['apple', 'banana', 'orange'];  
const hasBanana = fruits.includes('banana');
```

```
console.log(hasBanana); // true (banana가 배열에 존재함)
```

slice() 메소드

slice() 메소드는 배열의 일부분을 새로운 배열로 반환합니다.

javascript

```
const animals = ['ant', 'bison', 'camel', 'duck', 'elephant'];  
const sliced = animals.slice(1, 4);
```

```
console.log(sliced); // ['bison', 'camel', 'duck']
```

fill() 메소드 기본 사용법

javascript

// 배열의 모든 요소를 특정 값으로 채우기

```
const array1 = [1, 2, 3, 4, 5];  
array1.fill(0);  
console.log(array1); // [0, 0, 0, 0, 0]
```

// 시작 인덱스 지정하기

```
const array2 = [1, 2, 3, 4, 5];  
array2.fill(0, 2); // 인덱스 2부터 끝까지 0으로 채움  
console.log(array2); // [1, 2, 0, 0, 0]
```

// 시작 인덱스와 끝 인덱스 지정하기 (끝 인덱스는 포함되지 않음)

```
const array3 = [1, 2, 3, 4, 5];  
array3.fill(0, 1, 3); // 인덱스 1부터 3 이전까지(인덱스 1, 2)를 0으로 채움  
console.log(array3); // [1, 0, 0, 4, 5]
```

// 새 배열 만들고 채우기

```
const newArray = new Array(5).fill('A');  
console.log(newArray); // ['A', 'A', 'A', 'A', 'A']
```

sort() 메소드

sort() 메소드는 배열의 요소를 정렬합니다. 기본적으로 문자열로 변환 후 오름차순으로 정렬합니다.

javascript

```
const fruits = ['banana', 'apple', 'orange', 'grape'];  
fruits.sort();  
  
console.log(fruits); // ['apple', 'banana', 'grape', 'orange']
```

역정렬 (내림차순 정렬)

sort() 메소드에 비교 함수를 전달하여 역정렬(내림차순)을 구현할 수 있습니다.

```
const numbers = [1, 30, 4, 21, 100000];  
numbers.sort((a, b) => b - a); // 내림차순 정렬  
  
console.log(numbers); // [100000, 30, 21, 4, 1]
```

역정렬은 별도의 메소드가 있는 것이 아니라, sort() 메소드에 비교 함수를 사용하여 구현합니다. 문자열의 경우도 마찬가지로 비교 함수를 통해 내림차순 정렬이 가능합니다:

6. 배열과 문자열 변환

- ◆ join() - 배열을 문자열로 변환

```
let words = ["Hello", "world"];  
console.log(words.join(" ")); // "Hello world"
```

- ◆ split() - 문자열을 배열로 변환

javascript

코드 복사

```
let text = "apple,banana,grape";
```

```
let fruitArray = text.split(",");  
console.log(fruitArray); // ["apple", "banana", "grape"]
```

7. 배열 정렬

◆ 오름차순 정렬 (`sort()`)

```
let numbers = [4, 2, 7, 1];  
numbers.sort((a, b) => a - b);  
console.log(numbers); // [1, 2, 4, 7]
```

◆ 내림차순 정렬

```
numbers.sort((a, b) => b - a);  
console.log(numbers); // [7, 4, 2, 1]
```

정리

메서드	설명
<code>push()</code>	배열 끝에 요소 추가
<code>pop()</code>	배열 끝 요소 제거
<code>unshift()</code>	배열 앞에 요소 추가
<code>shift()</code>	배열 앞 요소 제거
<code>splice()</code>	특정 위치에 요소 추가/제거
<code>map()</code>	각 요소 변환 (새 배열 반환)
<code>filter()</code>	조건에 맞는 요소만 필터링
<code>reduce()</code>	배열 값을 하나로 축약
<code>sort()</code>	배열 정렬

✓ 배열을 다룰 때는 `map()`, `filter()`, `reduce()` 등의 고차 함수가 매우 유용합니다.



✓ JSON (JavaScript Object Notation) 기초 설명

JSON은 데이터를 저장하고 전송하기 위한 경량 데이터 형식입니다.

📌 **JavaScript**의 객체 표기법을 기반으로 하지만, 대부분의 프로그래밍 언어에서 사용 가능합니다.

📌 1. JSON의 특징

- ✓ 텍스트 기반 데이터 형식 (UTF-8 지원)
 - ✓ 가볍고 가독성이 좋음
 - ✓ 구조화된 데이터 저장 및 전송 가능
 - ✓ 프로그래밍 언어와 무관하게 사용 가능 (Java, JavaScript, Python, PHP 등에서 사용)
 - ✓ API 응답 형식으로 많이 사용됨 (REST API, AJAX, GraphQL 등)
-

2. JSON 문법

✓ JSON 데이터 구조

JSON은 키-값 쌍 (Key-Value Pair) 으로 구성됩니다.

- 객체 (Object) → {} 중괄호 사용
- 배열 (Array) → [] 대괄호 사용

✓ JSON 기본 예제

```
{  
  "name": "Alice",  
  "age": 25,  
  "isStudent": false,  
  "skills": ["JavaScript", "Python", "C++"],  
  "address": {  
    "city": "Seoul",  
    "zipCode": "12345"  
  }  
}
```

- ✓ 문자열은 항상 "(큰따옴표)"로 감싸야 함
 - ✓ 숫자, 불리언, 배열, 객체 등의 데이터 타입 지원
-

3. JSON 데이터 타입

JSON에서 사용할 수 있는 데이터 타입은 다음과 같습니다.

데이터 타입

예제

문자열 (String)	<code>"name": "Alice"</code>
숫자 (Number)	<code>"age": 25</code>
불리언 (Boolean)	<code>"isStudent": false</code>
배열 (Array)	<code>"skills": ["JavaScript", "Python"]</code>
객체 (Object)	<code>"address": { "city": "Seoul" }</code>
null 값	<code>"nickname": null</code>

4. JSON과 JavaScript 객체 변환

JavaScript 객체 → JSON 변환 (`JSON.stringify()`)

javascript

코드 복사

```
let person = {  
  name: "Alice",  
  age: 25,  
  isStudent: false  
};
```

```
let jsonString = JSON.stringify(person);  
console.log(jsonString);  
// '{"name":"Alice","age":25,"isStudent":false}' (문자열 형태)
```

`JSON.stringify()` 는 객체를 JSON 문자열로 변환

JSON → JavaScript 객체 변환 (`JSON.parse()`)

javascript

코드 복사

```
let jsonData = '{"name":"Alice","age":25,"isStudent":false}';  
  
let obj = JSON.parse(jsonData);  
console.log(obj.name); // Alice
```

```
console.log(obj.age); // 25
```

✓ `JSON.parse()` 는 JSON 문자열을 JavaScript 객체로 변환

JSON과 JavaScript 객체는 서로 비슷하지만 중요한 차이점이 있어요.

✓ 공통점

- 둘 다 키-값(**key-value**) 구조를 가지고 있음
- 데이터를 표현하는데 사용됨
- 중괄호 `{}`를 사용하여 객체를 나타냄

● 차이점

차이점	JSON	JavaScript 객체
형식	문자열(String)	객체(Object)
데이터 타입	문자열, 숫자, 불리언, 배열, null , 객체만 가능	함수, undefined , Symbol 등도 가능
속성 이름	항상 큰따옴표(" ")로 감싸야 함	큰따옴표 없이도 가능
사용 가능 여부	언어나 환경에 관계없이 사용 가능 (Python, Java, C 등)	JavaScript 에서만 사용 가능
메서드 포함 여부	값으로 메서드(함수) 저장 불가	가능

> 07. 이벤트 추가하기

1. 이벤트란? 웹 페이지에서 일어나는 사건(행동)을 감지하고 처리하는 시스템

- 예시:
 - 사용자가 버튼을 클릭
 - 마우스를 이동
 - 키보드를 누름
 - 페이지가 로드 완료됨

이런 사건(**Event**)을 감지해 원하는 동작을 실행합니다. 이때 동작들은 함수로 구현되고 이 함수를 이벤트 핸들러라고 한다.

1. 자바스크립트 간단 이벤트 처리 예제

```
<!DOCTYPE html>

<html lang="ko">

<head>

  <meta charset="UTF-8">

  <meta name="viewport" content="width=device-width, initial-scale=1.0">

  <title>Click 이벤트 예제</title>

</head>

<body>

  <h1>Click 이벤트 예제</h1>

  <button onclick="changeText()">클릭하세요</button>

  <p id="text">버튼을 클릭해 보세요!</p>

  <script>

    function changeText() {

      console.log("버튼이 클릭되었습니다.");

    }

  </script>

</body>

</html>
```

2. JSX에서의 클릭 이벤트 (onClick)

버튼을 클릭하면 `handleClick` 함수가 실행됩니다.

```
import React from "react";

function App() {
  const handleClick = () => {
    alert("버튼이 클릭되었습니다! 🎉");
  };

  return (
    <div>
      <h1>이벤트 기초 예제</h1>
      <button onClick={handleClick}>클릭하세요</button>
    </div>
  );
}

export default App;
```

✅ `onClick={handleClick}` → 버튼을 클릭할 때 `handleClick` 함수 실행

✅ `alert("버튼이 클릭되었습니다!")` → 클릭하면 팝업 메시지 표시

3. 고정된 형태의 클릭이벤트 추가되는 자식 컴포넌트 만들기

```

function ChildComponent() {

  const handleClick = () => {

    alert("ChildComponent Button clicked!"); // 고정된 기능 추가

  };

  return (

    <div>

      ChildComponent<br/>

      <button onClick={handleButtonClick}>클릭하세요</button>

    </div>

  );
}

```

고정 이벤트

```

import React from "react";

import ChildComponent from "../ChildComponent";

function App() {

  return (

    <div>

      <h1>부모 컴포넌트</h1>

      <ChildComponent />

      <ChildComponent />

    </div>

  );
}

```



```
    </div>

  );
}
```

```
export default App;
```

4. 그때 그때 이벤트 핸들러 변경하기

```
import React from 'react';
function ChildComponent(prop) {
  return (
    <div>
      {/* prop 객체를 통해 handleClick 함수에 접근합니다. */}
      <button onClick={prop.handleClick}>클릭하세요</button>
    </div>
  );
}
```

```
function App() {
  // 이벤트 핸들러
  const handleClick = () => {
    alert('버튼이 클릭되었습니다!');
  };
  const handleClick1 = () => {
    alert('handleClick1 이벤트 핸들러');
  };
  return (
    <div>
      {/* ChildComponent에 handleClick 함수를 prop으로 전달합니다. */}
      <ChildComponent handleClick={handleClick} />
      <ChildComponent handleClick={handleClick} />
      <ChildComponent handleClick={handleClick1} />
    <ChildComponent onClick={()=>{
      alert("arrow function")
    }} />
  );
}
```

```
}/>
```

```
</div>
```

```
);
```

```
}
```

> 08.hooks useState

리액트에는 여러 종류의 훅이 있는데 변수의 값이 변경되는 것을 감시하는 기능을 하는 것을 훅이라고 한다.

`useState`는 가장 대표적인 훅으로 변수의 값이 변경되면 화면을 다시 그려준다.

`useState`는 React에서 함수형 컴포넌트에서 ****상태(state)****를 관리할 수 있게 해주는 **Hook**입니다. 기본적인 문법은 다음과 같습니다:

```
const [state, setState] = useState(initialValue);
```

구성 요소 설명:

구성 요소	설명
<code>state</code>	현재 상태 값을 담고 있는 변수입니다.
<code>setState</code>	상태를 업데이트할 때 사용하는 함수입니다.
<code>initialValue</code>	상태의 초기값입니다. 문자열, 숫자, 객체, 배열, <code>boolean</code> 등 어떤 타입이든 가능

`state`가 적용 안되는 예제

```
import React from 'react';
```

```
const MyComponent = () => {
```

```

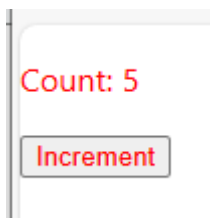
let count = 0; // useState를 사용하지 않음

const increment = () => {
  count += 1; // 값은 증가하지만, 리렌더링이 발생하지 않음
  console.log(count); // 콘솔에는 증가된 값이 보이지만, UI에는 반영되지 않음
};

return (
  <div>
    <p>Count: {count}</p> {/* UI가 업데이트되지 않음 */}
    <button onClick={increment}>Increment</button>
  </div>
);
};

export default MyComponent;

```



```

import React, { useState } from 'react';
const MyComponent = () => {
  // 상태 변수와 상태를 업데이트하는 함수 선언
  const [count, setCount] = useState(0);
  // 상태를 업데이트하는 함수
  const increment = () => {
    setCount(count + 1);
  };
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
};

```

```

        </div>
    );
};

function App() {

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

export default App;

```

사용방법

1. `useState`를 `import`한다.
2. `useState`를 선언한다. `const [count, setCount] = useState(0);`
3. `set`함수를 사용해서 값을 변경한다. `setCount(count + 1);`

배열 구조 분해 할당 (Destructuring Assignment)

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

이 코드에서 사용된 배열 구조 분해 할당(**Destructuring Assignment**) 문법을 설명하겠습니다.

1. 기본 개념

배열 구조 분해 할당은 배열의 요소를 변수에 쉽게 할당할 수 있도록 해주는 **JavaScript** 문법입니다.

예제

```
const arr = [1, 2];
const [a, b] = arr; // [a,b]=[b,a]

console.log(a); // 1
console.log(b); // 2
```

- `arr` 배열의 첫 번째 요소(1)가 `a`에, 두 번째 요소(2)가 `b`에 할당됩니다.

2. `useState`에서의 배열 구조 분해 할당

React의 `useState` 혹은 배열을 반환하는데, 이를 배열 구조 분해 할당을 이용하여 변수에 저장합니다.

javascript

복사편집

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

- `useState(0)`는 [현재 상태 값, 상태 변경 함수] 형태의 배열을 반환합니다.
- 배열 구조 분해 할당을 사용하여:
 - `count`는 현재 상태 값
 - `setCount`는 상태를 변경하는 함수로 저장됩니다.

이와 동일한 동작을 하는 코드 (배열 구조 분해 할당 없이)

javascript

복사편집

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

```
const stateArray = useState(0);
const count = stateArray[0];
const setCount = stateArray[1];
```

위처럼 `useState(0)`가 반환한 배열을 직접 변수에 저장할 수도 있지만, 배열 구조 분해 할당을 사용하면 훨씬 간결하게 코드를 작성할 수 있습니다.

3. 배열 구조 분해 할당의 특징

1) 필요한 값만 할당 가능

javascript

복사편집

```
const [count] = useState(0);  
console.log(count); // 현재 상태 값 (0)
```

- `setCount`를 사용하지 않을 경우 생략 가능하지만, 일반적으로 `setCount`도 함께 사용해야 하므로 생략하지 않는 것이 좋습니다.

2) 기본값 설정 가능

javascript

복사편집

```
const [a = 10, b = 20] = [];  
console.log(a); // 10  
console.log(b); // 20
```

- 빈 배열일 경우 기본값이 할당됩니다.

4. 정리

- 배열 구조 분해 할당은 배열의 요소를 개별 변수로 쉽게 저장할 수 있는 문법입니다.
- `useState`는 [현재 상태, 상태 변경 함수] 형태의 배열을 반환하므로, 배열 구조 분해 할당을 사용하면 편리합니다.
- `const [count, setCount] = useState(0);`는 `useState`의 반환값을 각각 `count`(현재 값)와 `setCount`(변경 함수)로 나누어 저장하는 코드입니다.

즉, 배열 구조 분해 할당을 이용하면 배열에서 원하는 값을 추출하여 가독성 좋고 간결한 코드를 작성할 수 있습니다! 🚀

0과 100으로 리셋할수 있는 카운터를 만들어 보자.

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

```
export default function Counter() {
  const [count, setCount] = useState(0);
  const increment = () => {
    if (count < 100) {
      setCount(count + 1);
    }
  };
  const decrement = () => {
    if (count > 0) {
      setCount(count - 1);
    }
  };
  const reset = () => {
    setCount(0);
  };
  const setToMax = () => {
    setCount(100);
  };
  return (
    <div>
      <h2>카운터</h2>
      <div>
        <button onClick={decrement}>-</button>
        <span>{count}</span>
        <button onClick={increment}>+</button>
      </div>
      <div>
        <button onClick={reset}>0으로 초기화</button>
        <button onClick={setToMax}>최대값(100)</button>
      </div>
      <div>
        현재 값: {count}
      </div>
    </div>
  );
}
```

> 08. 다양한 이벤트

◆ 2. 입력 필드 변경 이벤트 (onChange)

사용자가 입력하면, 입력한 값을 실시간으로 화면에 표시하는 예제입니다.

```
import React, { useState } from "react";

function App() {
  const [text, setText] = useState("");

  const handleChange = (event) => {
    setText(event.target.value);
  };

  return (
    <div>
      <h1>입력 필드 이벤트</h1>
      <input type="text" onChange={handleChange} placeholder="입력하세요" />
      <p>입력한 값: {text}</p>
    </div>
  );
}

export default App;
```

- ✓ `useState`를 사용하여 입력값(`text`)을 저장
- ✓ `onChange={handleChange}` → 입력할 때마다 `handleChange` 실행
- ✓ `event.target.value` → 사용자가 입력한 값을 가져와 `text`에 저장

자식 컴포넌트 `ChangeApp`로 만들기

```
import React, { useState } from "react";

function ChangeApp() {

  const [text, setText] = useState("");

  const handleChange = (event) => {

    setText(event.target.value);

  };

  return (

    <div>

      <h1>입력 필드 이벤트</h1>

      <input type="text" onChange={handleChange} placeholder="입력하세요" />

      <p>입력한 값: {text}</p>

    </div>

  );

}
```

```
function App() {

  return (

    <div>

      <ChangeApp></ChangeApp>

      <ChangeApp></ChangeApp>

    </div>

  );
}

export default App;
```

컴포넌트 마다 이벤트가 다르게 적용하게 해보자.

myFunction 와 **myFunction()**의 차이

```
function myFunction(){

  return "functionString";

}
```

myFunction은 함수가 필요할때

myFunction()은 함수의 리턴값이 필요할때

```
var a= myFunction;

console.log(a); //함수 내용

console.log(a()); //functionString
```

```
var a=myFunction();

console.log(a);//functionString
```

```
import React, { useState } from "react";

function ChangeApp(props) {

  const [text, setText] = useState("");

  const handleChange = (event) => {

    //공동 작업 추가

    setText(props.onChange());//사용자가 원하는 작업 추가

    //공동 작업 추가

  };

  return (

    <div>

      <h1>입력 필드 이벤트</h1>

      <input type="text" onChange={handleChange} placeholder="입력하세요" />

      <input type="text" onChange={props.onChange} placeholder="입력하세요" />

      <p>입력한 값: {text}</p>

    </div>

  );

}

function App() {

  let counter=0;

  return (
```

```

    <div>

      <ChangeApp onChange={ (event) => {

        return counter++;

      }}></ChangeApp>

    </div>

  );
}

```

```
export default App;
```

◆ 3. 마우스 이벤트 (onMouseEnter, onMouseLeave)

마우스를 올리거나 벗어날 때 배경색이 변하는 예제입니다.

jsx

복사편집

```

import React, { useState } from "react";

function App() {
  const [bgColor, setBgColor] = useState("white");

  return (
    <div>
      <h1>마우스 이벤트</h1>
      <div
        style={{ width: "200px", height: "100px", backgroundColor:
bgColor, textAlign: "center", lineHeight: "100px" }}
        onMouseEnter={() => setBgColor("lightblue")}
        onMouseLeave={() => setBgColor("white")}

```

```

    >
      마우스를 올려보세요!
    </div>
  </div>
);
}

```

```
export default App;
```

✓ `onMouseEnter` → 마우스를 올리면 배경색 변경

✓ `onMouseLeave` → 마우스를 벗어나면 원래 색으로 변경

◆ 4. 키보드 이벤트 (`onKeyDown`, `onKeyUp`)

사용자가 키보드를 누를 때, 어떤 키를 눌렀는지 출력하는 예제입니다.

jsx

복사편집

```
import React, { useState } from "react";
```

```
function App() {
  const [key, setKey] = useState("");

```

```

  const handleKeyDown = (event) => {
    setKey(event.key);
  };

```

```

  return (
    <div>
      <h1>키보드 이벤트</h1>
      <input type="text" onKeyDown={handleKeyDown} placeholder="키를
눌러보세요" />
      <p>입력한 키: {key}</p>
    </div>
  );

```

```
}
```

```
export default App;
```

✓ `onKeyDown` → 사용자가 키를 누를 때 `handleKeyDown` 실행

✓ `event.key` → 누른 키 값을 가져와 `key` 상태에 저장

◆ 5. 폼 제출 이벤트 (`onSubmit`)

폼을 제출할 때 입력한 데이터를 확인하는 예제입니다.

```
import React, { useState } from "react";
```

```
function App() {
```

```
  const [name, setName] = useState("");
```

```
  const handleSubmit = (event) => {
```

```
    event.preventDefault(); // 기본 폼 제출 동작 방지
```

```
    alert(`제출된 이름: ${name}`);
```

```
  };
```

```
  return (
```

```
    <div>
```

```
      <h1>폼 제출 이벤트</h1>
```

```
      <form onSubmit={handleSubmit}>
```

```
        <input type="text" value={name} onChange={(e) =>
```

```
setName(e.target.value)} placeholder="이름 입력" />
```

```
        <button type="submit">제출</button>
```

```
      </form>
```

```
    </div>
```

```
  );
```

```
}
```

```
export default App;
```

- ✓ `event.preventDefault()` → 페이지 새로고침 방지
- ✓ `onSubmit={handleSubmit}` → 제출 버튼 클릭 시 실행

🎯 정리

이벤트 종류	설명	예제
<code>onClick</code>	클릭 시 실행	버튼 클릭
<code>onChange</code>	입력값 변경 시 실행	텍스트 입력 필드
<code>onMouseEnter</code> / <code>onMouseLeave</code>	마우스 올릴 때 / 벗어날 때 실행	배경색 변경
<code>onKeyDown</code> / <code>onKeyUp</code>	키보드 누를 때 / 뗄 때 실행	키 입력 감지
<code>onSubmit</code>	폼 제출 시 실행	폼 데이터 확인

이제 `React`에서 이벤트를 쉽게 다룰 수 있을 거예요! 🎉

```
import React, { useState } from "react";

// 폼 컴포넌트
function FormComponent({ name, setName, handleSubmit }) {
  return (
    <div>
      <h1>폼 제출 이벤트</h1>
      <form onSubmit={handleSubmit}>
        <input
          type="text"
          value={name}
          onChange={(e) => setName(e.target.value)}
          placeholder="이름 입력"
        />
        <button type="submit">제출</button>
      </form>
    </div>
  );
}
```



```

    );
}

// 메인 컴포넌트
function App() {
    const [name, setName] = useState("");

    const handleSubmit = (event) => {
        event.preventDefault();
        alert(`제출된 이름: ${name}`);
    };

    return (
        <div>
            <FormComponent name={name} setName={setName} handleSubmit={handleSubmit}
        />
        </div>
    );
}

export default App;

```

전통적인 방식:

```

javascript
function FormComponent(props) {
    const name = props.name;
    const setName = props.setName;
    const handleSubmit = props.handleSubmit;

    // 함수 내용...
}

```

구조분해 할당 방식:

```

javascript
function FormComponent({ name, setName, handleSubmit }) {

```

```
// 함수 내용...  
}
```

구조 분해 할당 (Destructuring Assignment) 초간단 설명

배열이나 객체의 내용을 일반 변수에 넣는작업

👉 배열이나 객체의 값을 변수로 "풀어서" 할당하는 문법

1. 배열 구조 분해

기본 사용법

```
const arr = [1, 2, 3];  
  
// 배열의 값을 순서대로 변수에 할당  
const [a, b, c] = arr;  
  
console.log(a); // 1  
console.log(b); // 2  
console.log(c); // 3
```

- `[a, b, c] = arr`의 순서대로 값이 분해되어 할당됩니다.

기능 추가 설명

(1) 필요한 값만 가져오기 (건너뛰기)

```
const [x, , z] = [10, 20, 30];  
console.log(x); // 10  
console.log(z); // 30 (20은 무시됨)
```

- `,`로 빈 칸을 두면 해당 인덱스는 건너됩니다.

(2) 기본값 설정 (값이 없을 때)

```
const [name = "Guest", age = 20] = ["Tom"];  
console.log(name); // "Tom" (배열에 값이 있음)
```

```
console.log(age); // 20 (배열에 없으므로 기본값 사용)
```

- 변수 = 기본값 형태로 값이 없을 때 대체할 기본값을 지정할 수 있습니다.

(3) 나머지 값 한 번에 가져오기 (...)

```
const [first, ...rest] = [1, 2, 3, 4];  
console.log(first); // 1  
console.log(rest); // [2, 3, 4] (나머지가 배열로 할당)
```

- ...rest로 남은 요소들을 배열로 저장할 수 있습니다.
-

2. 객체 구조 분해

기본 사용법

```
const user = { name: "Alice", age: 25 };
```

```
// 객체의 키 이름으로 변수 할당
```

```
const { name, age } = user;
```

```
console.log(name); // "Alice"
```

```
console.log(age); // 25
```

- { name, age } = user 객체에서 키 이름과 동일한 변수로 값을 분해합니다.

기능 추가 설명

(1) 변수명 변경하기 (새변수: 원래키)

```
const { name: userName, age: userAge } = user;  
console.log(userName); // "Alice" (name → userName으로 변경)
```

- 키: 새변수명으로 다른 이름의 변수에 할당할 수 있습니다.

(2) 기본값 설정 (키가 없을 때)

```
const { name = "Guest", isAdmin = false } = { name: "Bob" };  
console.log(name); // "Bob" (객체에 값이 있음)  
console.log(isAdmin); // false (객체에 없으므로 기본값 사용)
```

- 키 = 기본값으로 값이 없을 때의 기본값을 지정합니다.

(3) 중첩 객체 분해

```
const user = {  
  id: 1,  
  info: { email: "alice@example.com", phone: "010-1234-5678" }  
};
```

```
const { info: { email, phone } } = user;  
console.log(email); // "alice@example.com"
```

- `{ 키: { 내부키 } }` 형태로 중첩된 객체의 값을 분해할 수 있습니다.

💡 구조 분해 할당의 장점

1. 코드가 간결해집니다.

```
// 기존 방식  
const firstName = user.firstName;  
const lastName = user.lastName;
```

```
// 구조 분해 할당  
const { firstName, lastName } = user;
```

2. 함수에서 객체/배열 인자 쉽게 받기

```
function printUser({ name, age }) {  
  console.log(`${name} (${age}세)`);  
}  
printUser({ name: "Tom", age: 30 }); // "Tom (30세)"
```

🚀 초간단 요약

분류	예시	설명
배열	<code>const [a, b] = [1, 2]</code>	순서대로 변수 할당
객체	<code>const { name, age } = user</code>	키 이름으로 변수 할당

기본값

```
const { x = 10 } = {}
```

값이 없을 때 기본값 적용

변수명 변경

```
const { name: userName } = user
```

`name` → `userName`으로 저장

➡ "배열/객체의 값을 편리하게 변수로 풀어 쓸 때 사용!"


```

        <button
          onClick={() => handleAnswer("X")}
          style={{ padding: "10px 20px", fontSize: "18px" }}
        >
          X
        </button>
      </>
    ) : (
      <div>
        <h2>게임 종료!</h2>
        <p>당신의 점수: {score} / {quizData.length}</p>
        <button onClick={() => { setCurrentIndex(0); setScore(0);
setShowResult(false); }}>
          다시 시작 ↺
        </button>
      </div>
    )}
  </div>
);
}

export default App;

```

1. 주제별 OX 퀴즈 앱

- 기능: 수학/과학/역사 등 카테고리 선택
- 기술: `<select>` 드롭다운, `filter` 사용

✓ 2. 난이도 조절 기능

- 기능: 쉬움/보통/어려움 선택 시 문제 난이도 달라짐
- 기술: 조건 분기 렌더링, 배열 필터링

✓ 3. 타이머 기반 제한 시간 퀴즈

- 기능: 문제당 10초 제한 → 지나면 자동 오답 처리
- 기술: `useEffect`, `setTimeout`, `clearTimeout`

✓ 4. 이미지 기반 퀴즈

- 기능: 텍스트 대신 이미지 문제 (예: 동물 사진)
- 기술: `` 사용, 상태로 이미지 관리

✓ 5. 점수 누적 + 리더보드 저장

- 기능: 이름 + 점수 저장, 최고 기록 표시
- 기술: `localStorage`, 정렬, JSON 저장

✓ 6. 문제 추가/수정/삭제 (관리자 모드)

- 기능: 폼으로 문제 입력, 삭제/수정 가능
- 기술: `useState`, 리스트 관리, CRUD 개념

✓ 7. 2인 플레이 모드

- 기능: Player1 → Player2 번갈아 문제 풀이
- 기술: 두 개의 점수 상태 관리, 턴 전환

✓ 8. 퀴즈 앱 라우팅 구조 분리

- 기능: `/`, `/quiz`, `/result`, `/admin` 페이지

- 기술: **React Router**, SPA 구조 학습
-

✓ 9. 오답 복습 모드

- 기능: 틀린 문제만 다시 풀기
 - 기술: 배열 필터링, 상태 분기
-

✓ 10. 퀴즈 결과 공유 (복사 or SNS)

- 기능: "결과 복사", SNS 공유
 - 기술: **navigator.clipboard**, 공유 API
-

✓ 11. 정답 해설 기능

- 기능: 문제마다 정답 이유 출력
 - 기술: 문제 객체에 **explanation** 추가
-

✓ 12. 테마 변경 (다크모드/라이트모드)

- 기능: 테마 버튼 → 스타일 변경
 - 기술: **CSS** 조건 렌더링, 상태 기반 클래스 변경
-

✓ 13. 퀴즈 즐겨찾기 기능

- 기능: 문제에 즐겨찾기(★) 표시 및 저장
 - 기술: 배열 상태 관리, **localStorage**
-

✓ 14. 무작위 퀴즈 모드 (랜덤 순서)

- 기능: 문제를 무작위로 섞어 제공
 - 기술: `Array.sort(() => Math.random() - 0.5)`
-

✓ 15. 퀴즈 카운트다운 시작 애니메이션

- 기능: 시작 전에 “3...2...1...” 카운트다운
 - 기술: `setInterval`, 간단한 애니메이션 효과
-

✓ 16. 배경음악 + 효과음

- 기능: 버튼 클릭 시 효과음, 배경 음악
 - 기술: `Audio API`, `<audio>` 태그
-

✓ 17. 성취 뱃지/레벨 시스템

- 기능: 점수에 따라 “OX 마스터” 등 뱃지 지급
 - 기술: 조건 기반 UI, 상태 기반 뱃지 렌더링
-

✓ 18. 결과 분석 차트

- 기능: 맞힌 문제/틀린 문제 수를 그래프로 표현
 - 기술: `Chart.js` 또는 `Recharts` 사용
-

✓ 19. 클라우드 연동 리더보드

- 기능: `Firebase`와 연동해 전 세계 사용자 점수 저장
- 기술: `Firebase Firestore`, 인증, 실시간 DB

✓ 20. 영어/한글 등 다국어 지원

- 기능: UI 및 문제를 여러 언어로 전환
- 기술: 다국어 JSON 관리, **i18n** 기본 개념

스프레드 연산자 (...)란?

****스프레드 연산자 (...)****는 배열이나 객체의 요소를 개별적으로 펼치는(**spread**) 역할을 합니다. 즉, 기존 값을 복사하여 새로운 배열이나 객체를 쉽게 만들 수 있습니다.

1. 배열에서 ... 스프레드 연산자 사용

배열의 요소를 개별적으로 펼쳐서 새로운 배열을 만들 수 있습니다.

✓ 배열 복사

javascript

복사편집

```
const students = ["Alice", "Bob", "Charlie"];
const newStudents = [...students];
const newStudents2=students;
```

```
console.log(newStudents); // ["Alice", "Bob", "Charlie"]
console.log(newStudents2); // ["Alice", "Bob", "Charlie"]
```

newStudents랑 newStudents2의 차이는

newStudents2는 =를 사용해서 students랑 같은 주소를 가지고 있다.

newStudents랑 스프레드 연산자를 사용해서 students는 다른 주소를 가지고 있다.

📌 설명

- `...students` → `["Alice", "Bob", "Charlie"]`가 펼쳐짐
 - `newStudents`는 `students`의 복사본이므로 원본을 변경해도 영향을 받지 않음
-

✓ 배열에 새로운 요소 추가

javascript

복사편집

```
const students = ["Alice", "Bob"];
const newStudent = "Charlie";
```

// 기존 배열에 새로운 요소 추가

```
const newStudents = [...students, newStudent];
console.log(newStudents); // ["Alice", "Bob", "Charlie"]
```

`students.push("Charlie");` //새로 배열이 만들어진것이 아니고 기존 배열에 추가

```
const [students,setStudents]=useState(["Alice", "Bob"]);
setStudents([...students, "Charlie"]);//화면갱신이 일어 난다.
```

```
students.push("Charlie");
setStudents(students);//화면갱신이 안일어 난다.
```

📌 설명

- `...students` → 기존 배열의 요소(`"Alice", "Bob"`)를 펼침
- `newStudent("Charlie")`를 추가하여 새로운 배열을 생성

- 원본 배열(**students**)은 변경되지 않음
-

✓ 배열 합치기

```
const groupA = ["Alice", "Bob"];
const groupB = ["Charlie", "David"];

const allStudents = [...groupA, ...groupB];

console.log(allStudents); // ["Alice", "Bob", "Charlie", "David"]
```

📌 설명

- **...groupA**와 **...groupB**를 펼쳐서 하나의 배열로 합침
-

2. 객체에서 ... 스프레드 연산자 사용

객체의 속성을 복사하거나 새로운 속성을 추가할 때 사용합니다.

✓ 객체 복사

```
const student = { name: "Alice", age: 20 };
const copiedStudent = { ...student };

console.log(copiedStudent); // { name: "Alice", age: 20 }
```

📌 설명

- **...student**를 사용하여 새로운 객체를 생성
 - 원본(**student**)을 변경해도 **copiedStudent**에는 영향 없음
-

✓ 객체 속성 추가 및 업데이트

```
const student = { name: "Alice", age: 20 };

// 기존 객체 복사 + 새로운 속성 추가
const updatedStudent = { ...student, grade: "A" };

console.log(updatedStudent); // { name: "Alice", age: 20, grade: "A" }
```

설명

- 기존 객체(`student`)의 속성을 유지하면서 `grade` 속성을 추가

```
const student = { name: "Alice", age: 20 };

// 기존 속성 덮어쓰기
const updatedStudent = { ...student, age: 21 };


console.log(updatedStudent); // { name: "Alice", age: 21 }
```

설명

- `age: 21`이 `student`의 기존 `age: 20`을 덮어씀

핵심 정리

- ✓ 배열에서 `...` → 배열 요소를 펼쳐서 복사, 추가, 합치기 가능
- ✓ 객체에서 `...` → 객체의 복사, 새로운 속성 추가 및 업데이트 가능
- ✓ 불변성 유지 → 원본 데이터를 직접 수정하지 않고 새로운 데이터 생성 가능

✅ `...` 스프레드 연산자는 코드의 가독성과 유지보수성을 높이는 중요한 기능! 

가장 간단한 조건부 렌더링 예제

```
<button onclick="showText()">눌러보세요</button>
```

```
<p id="message"></p>
```

`<script>`

```
let visible = false;
```

```
function showText() {
```

```
    visible = !visible; // true ↔ false 전환
```

```
    // 조건부 렌더링
```

```
    document.getElementById("message").innerHTML =
```

```
        visible && "안녕하세요!";
```

```
}
```

`</script>`

동작 설명:

1. 처음에는 빈 화면 (`visible = false`)
2. 버튼 클릭시:
 - `visible`이 `true`로 바뀜 → "안녕하세요!" 표시
 - 다시 클릭하면 `visible`이 `false`로 바뀜 → 텍스트 사라짐

핵심 원리:

- 조건 `&&` 내용 → 조건이 `true`일 때만 내용이 실행됨
- `false && "안녕하세요!"` → 아무것도 표시 안됨
- `true && "안녕하세요!"` → "안녕하세요!" 표시

```
function App() {
```

```
    const isShow = true; // 조건 변수 (true/false 변경해보세요)
```

```
    return (
```

```
        <div>
```

```
{isShow && <p>이 문구는 조건이 true일 때만 보입니다!</p>}
```

```
</div>
```

```
);
```

```
}
```

```
export default App;
```

```
import React, { useState } from "react";
```



```

function App() {
  const [students, setStudents] = useState([
    { id: 1, name: "Alice", age: 21 },
    { id: 2, name: "Bob", age: 22 },
    { id: 3, name: "Charlie", age: 23 },
  ]);
  const [selectedStudent, setSelectedStudent] = useState(null);
  const [newStudent, setNewStudent] = useState({ name: "", age: "" });

  return (
    <div style={{ padding: "20px" }}>
      <h1>학생 목록</h1>
      <ul>
        {students.map((student) => (
          <li key={student.id}>
            <button onClick={() => setSelectedStudent(student)}>
              {student.name}
            </button>
            age : {student.age}

          </li>
        ))}
      </ul>

      {selectedStudent && (
        <div style={{ marginTop: "20px", border: "1px solid black", padding: "10px" }}>
          <h2>학생 정보</h2>
          <p>이름: {selectedStudent.name}</p>
          <p>나이: {selectedStudent.age}세</p>
        </div>
      )}

      <div style={{ marginTop: "20px" }}>
        <h2>새 학생 추가</h2>
        <input
          type="text"
          placeholder="이름"
          value={newStudent.name}
          onChange={(e) => setNewStudent({ ...newStudent, name: e.target.value })}
        />
        <input

```

```

        type="number"
        placeholder="나이"
        value={newStudent.age}
        onChange={(e) => setNewStudent({ ...newStudent, age: e.target.value })}
      />
    <button
      onClick={() => {
        const newId = students.length + 1;
        setStudents([...students, { id: newId, name: newStudent.name, age:
newStudent.age }]);
        setNewStudent({ name: "", age: "" });
      }}
    >
      추가
    </button>
  </div>
</div>
);
}

export default App;

```

다음 예제의 변화된 예제와 차이를 확인하자.
 다음 예제는 입력하면 text창의 text가 변경된다.

```

<input
  type="text"
  placeholder="이름"
  value={newStudent.name}
  onChange={(e) => setNewStudent({ ...newStudent, name: e.target.value })}
/>

```

다음예제는 입력치의 값이 고정되어 있어서 입력해도 입력창에 입력되지 않는다.

```
<input
  type="text"
  placeholder="이름"
  value={newStudent.name}
/>
```

다음예제는 value값이 고정되어 있지 않아서 text창을 통해서 변경된다.

```
<input
  type="text"
  placeholder="이름"

  onChange={(e) => setNewStudent({ ...newStudent, name: e.target.value })}
/>
```

> 15. 단어장 만들기

> 16.

> 17. 쉽게 개선선

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

import './App.css';

//학생 정보 시스템 메인 컴포넌트

const StudentApp=()=>{

  // 현재 모드 상태 관리(현재 화면 상태) (HOME, CREATE, SELECT, UPDATE, DELETE)

  const [mode, setMode] = useState('HOME');

  // 학생 데이터 상태 관리 (초기 데이터 4명 설정)

  const [students, setStudents] = useState([

    { id: 1, name: 'Alice', username: 'alice123', age: 21, height: 160, joinDate: '2020-01-01' },

    { id: 2, name: 'Bob', username: 'bob123', age: 22, height: 170, joinDate: '2019-03-15' },

    { id: 3, name: 'Charlie', username: 'charlie123', age: 23, height: 180, joinDate: '2018-05-10' },

    { id: 4, name: 'Dave', username: 'dave123', age: 24, height: 175, joinDate: '2017-07-20' },

  ]);

  //선택된 학생 ID 관리

  const [selectedId, setSelectedId] = useState(null);

  //다음 학생 ID 생성을 위한 상태 관리

  const [nextId, setNextId] = useState(5);

  //선택된 학생 정보 찾기

  const selectedStudent = students.find(s => s.id === selectedId);
```

//학생 생성 핸들러

```
const handleCreate = (student) => {  
  
  var newStudents=[...students, { ...student, id: nextId }];  
  
  setStudents(newStudents);  
  
  setNextId(nextId + 1);  
  
  setMode('SELECT');  
  
  console.log(newStudents);  
  
  //아래와 같이 기술하면 비동기 처리되어 딜레이가 생긴다.  
  
  //setStudents([...students, { ...student, id: nextId }]);  
};
```

//학생 정보 업데이트 핸들러

```
const handleUpdate = (updatedStudent) => {  
  
  setStudents(students.map(s => s.id === selectedId ? { ...s, ...updatedStudent }  
: s));  
  
  setMode('SELECT');  
};
```

//학생 삭제 핸들러

```
const handleDelete = () => {  
  
  if (selectedId) {  
  
    setStudents(students.filter(s => s.id !== selectedId));  
  
    setSelectedId(null);  
  
    setMode('SELECT');  
  
  }  
};
```

```

return(

  <div className="app-container">

    <header>

      <h1>Student Info System</h1>

      <div className="menu-buttons">

        <button onClick={() => { setMode('CREATE'); setSelectedId(null);
}}>CREATE</button>

        <button onClick={() => { setMode('SELECT'); setSelectedId(null);
}}>SELECT</button>

        <button onClick={() => { selectedId ? setMode('UPDATE') : alert('수정할
학생을 선택하세요'); }}>UPDATE</button>

        <button onClick={() => { selectedId ? setMode('DELETE') : alert('삭제할
학생을 선택하세요'); }}>DELETE</button>

      </div>

      <p className="mode-indicator">현재 페이지: {mode}</p>

    </header>

    <main>

      {mode === 'HOME' && <p>메뉴를 선택해주세요.</p>}

      {mode === 'CREATE' && <StudentCreate onCreate={handleCreate} />}

      {mode === 'SELECT' && (

        <>

          <StudentSelect students={students} onSelect={id => setSelectedId(id)}

          />

          {selectedStudent ? (

            <StudentDetail student={selectedStudent} />


```

```

    ) : (
      <p>학생을 선택하세요.</p>
    )}
  </>
)}

{mode === 'UPDATE' &&
  (selectedStudent ? <StudentUpdate student={selectedStudent}
onUpdate={handleUpdate} /> : <p>학생을 선택하세요.</p>)
}

{mode === 'DELETE' &&
  <StudentDelete
    student={selectedStudent}
    onDelete={handleDelete}
    onCancel={() => setMode('SELECT')}
  />
}

</main>

</div>

);

};

```

```

const StudentDelete = ({ student, onDelete, onCancel }) => {
  if (!student) return <p>학생을 선택하세요.</p>;
  return (
    <div className="student-delete">
      <p><strong>{student.name}</strong> 학생을 삭제하시겠습니까?</p>
      <button onClick={onDelete}>삭제</button>
    </div>
  );
};

```



```

        <button onClick={onCancel}>취소</button>

    </div>

);

};

const StudentUpdate = ({ student, onUpdate }) => {

    const [form, setForm] = useState({ ...student });

    const handleChange = (e) => {

        const { name, value } = e.target;

        setForm(prev => ({ ...prev, [name]: value }));

    };

    const handleSubmit = (e) => {

        e.preventDefault();

        onUpdate(form);

    };

    return (

        <form onSubmit={handleSubmit}>

            <h2>학생 수정</h2>

            <div><input name="name" placeholder="이름" value={form.name}
onChange={handleChange} required /></div>

            <div><input name="username" placeholder="아이디" value={form.username}
onChange={handleChange} required /></div>

            <div><input name="age" type="number" placeholder="나이" value={form.age}
onChange={handleChange} required /></div>

            <div><input name="height" type="number" placeholder="키(cm)"
value={form.height} onChange={handleChange} required /></div>

```

```
value={form.joinDate} onChange={handleChange} required /></div>
```

<button type="submit">수정 완료</button>

</form>

);

};

```
const StudentSelect = ({ students, onSelect }) => (
```

```
<div className="student-list">
```

<h2>학생 목록</h2>

|
 ID | 이름 | 아ㅣ0ㅣ□ㅣ</th> | 나이 | ၁ |

<th>가입일</th>

<th>선택</th>

</thead>

<tbody>

```
{students.map(student => (
```

```
<tr key={student.id}>
```

 {student.id} |

```
<td>{student.name}</td>
```

```

        <td>{student.username}</td>

        <td>{student.age}</td>

        <td>{student.height}cm</td>

        <td>{student.joinDate}</td>

        <td>

            <button onClick={() => onSelect(student.id)}>선택</button>

        </td>

    </tr>

    )))

</tbody>

</table>

</div>

);

```

```

const StudentDetail = ({ student }) => (

    <div className="student-detail">

        <h2>학생 상세보기</h2>

        <p><strong>이름:</strong> {student.name}</p>

        <p><strong>아이디:</strong> {student.username}</p>

        <p><strong>나이:</strong> {student.age}</p>

        <p><strong>키:</strong> {student.height}cm</p>

        <p><strong>가입일:</strong> {student.joinDate}</p>

    </div>

);

```

```

const StudentCreate = ({ onCreate }) => {

  const [student, setStudent] = useState({ name: '', username: '', age: '', height:
'', joinDate: '' });

  const handleChange = (e) => {

    const { name, value } = e.target;

    setStudent(prev => ({ ...prev, [name]: value }));

    //setStudent다음에 콜백함수(매개변수가 함수인 인자)가 오면

    //prev 변수에 이전 student값을 가져오고 함수의 리턴값이 새로운 변수가 된다.

  };

  const handleSubmit = (e) => {

    e.preventDefault();

    onCreate(student);

    setStudent({ name: '', username: '', age: '', height: '', joinDate: '' });

  };

  return (

    <form onSubmit={handleSubmit}>

      <h2>학생 추가</h2>

      <div><input name="name" placeholder="이름" value={student.name}
onChange={handleChange} required /></div>

      <div><input name="username" placeholder="아이디" value={student.username}
onChange={handleChange} required /></div>

      <div><input name="age" type="number" placeholder="나이" value={student.age}
onChange={handleChange} required /></div>

      <div><input name="height" type="number" placeholder="키(cm)"
value={student.height} onChange={handleChange} required /></div>

```

```
    <div><input name="joinDate" type="date" placeholder="가입일"
value={student.joinDate} onChange={handleChange} required /></div>

    <button type="submit">등록</button>

  </form>

);

};
```

```
const App = () => {

  return (

    <>

      <StudentApp></StudentApp>

    </>

  );

};
```

```
export default App;
```

```
//css부분
```

```
.App {  
  text-align: center;  
}
```

```
.App-logo {  
  height: 40vmin;  
  pointer-events: none;  
}
```

```
@media (prefers-reduced-motion: no-preference) {  
  .App-logo {  
    animation: App-logo-spin infinite 20s linear;  
  }  
}
```

```
.App-header {  
  background-color: #282c34;  
  min-height: 100vh;  
  display: flex;  
  flex-direction: column;  
  align-items: center;
```

```
    justify-content: center;

    font-size: calc(10px + 2vmin);

    color: white;
}
```

```
.App-link {

    color: #61dafb;
}
```

```
@keyframes App-logo-spin {

    from {

        transform: rotate(0deg);

    }

    to {

        transform: rotate(360deg);

    }

}
```

```
body {

    font-family: 'Noto Sans KR', sans-serif;

    background: #f5f7fa;

    margin: 0;

    padding: 0;
}
```

```
.container {

    width: 90%;
}
```

```
    max-width: 800px;

    margin: 0 auto;

    padding: 20px;
}
```

```
.header {

    text-align: center;

    margin-bottom: 20px;
}
```

```
.mode-navigation {

    display: flex;

    justify-content: center;

    gap: 10px;

    margin-bottom: 20px;
}
```

```
.mode-navigation button {

    padding: 10px 15px;

    font-size: 14px;

    background-color: #0066cc;

    color: white;

    border: none;

    border-radius: 6px;

    cursor: pointer;
}
```



```
.mode-navigation button:hover {  
    background-color: #005bb5;  
}
```

```
.current-mode {  
    text-align: center;  
    margin-bottom: 20px;  
}
```

```
.error {  
    color: red;  
    font-weight: bold;  
}
```

```
.form, .list, .detail, .delete-confirm {  
    background: white;  
    padding: 20px;  
    border-radius: 8px;  
    box-shadow: 0 2px 6px rgba(0,0,0,0.1);  
}
```

```
.form input {  
    display: block;  
    width: 100%;  
    margin-bottom: 15px;
```

```
padding: 8px;

font-size: 14px;

}
```

```
.form button {

width: 100%;

padding: 10px;

background-color: #28a745;

border: none;

color: white;

border-radius: 6px;

font-size: 16px;

cursor: pointer;

}
```

```
.form button:hover {

background-color: #218838;

}
```

```
.list-item {

margin-bottom: 10px;

}
```

```
.list-item button {

width: 100%;

padding: 10px;
```

```
background-color: #f1f1f1;

border: 1px solid #ccc;

border-radius: 6px;

cursor: pointer;

}
```

```
.list-item.selected button {

background-color: #007bff;

color: white;

}
```

```
.danger {

background-color: #dc3545;

margin-right: 10px;

}
```

```
.danger:hover {

background-color: #c82333;

}
```

```
/* App.css */
```

```
body {

margin: 0;

padding: 0;

font-family: 'Noto Sans KR', sans-serif;

background-color: #f8f9fa;
```

```
}
```

```
.app-container {  
  max-width: 900px;  
  margin: 40px auto;  
  padding: 20px;  
  background: white;  
  box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1);  
  border-radius: 12px;  
}
```

```
header {  
  text-align: center;  
  margin-bottom: 30px;  
}
```

```
header h1 {  
  margin-bottom: 10px;  
  font-size: 32px;  
  color: #343a40;  
}
```

```
.menu-buttons {  
  margin-bottom: 15px;  
}
```

```
.menu-buttons button {  
  
  margin: 0 8px;  
  
  padding: 10px 20px;  
  
  background-color: #6c5ce7;  
  
  color: white;  
  
  border: none;  
  
  border-radius: 8px;  
  
  font-size: 16px;  
  
  cursor: pointer;  
  
  transition: background-color 0.3s;  
  
}
```

```
.menu-buttons button:hover {  
  
  background-color: #5a4fcf;  
  
}
```

```
.mode-indicator {  
  
  margin-top: 10px;  
  
  font-size: 18px;  
  
  color: #555;  
  
}
```

```
main {  
  
  margin-top: 20px;  
  
}
```

```
form {  
  
  display: flex;  
  
  flex-direction: column;  
  
  gap: 12px;  
  
}
```

```
form input {  
  
  padding: 10px;  
  
  font-size: 16px;  
  
  border: 1px solid #ccc;  
  
  border-radius: 8px;  
  
}
```

```
form button {  
  
  padding: 10px;  
  
  background-color: #00b894;  
  
  color: white;  
  
  font-size: 16px;  
  
  border: none;  
  
  border-radius: 8px;  
  
  cursor: pointer;  
  
  transition: background-color 0.3s;  
  
}
```

```
form button:hover {  
  
  background-color: #019875;  
  
}
```

```
}
```

```
.student-list {  
    margin-top: 20px;  
}
```

```
.student-list table {  
    width: 100%;  
    border-collapse: collapse;  
    background: #fff;  
    border-radius: 12px;  
    overflow: hidden;  
    box-shadow: 0 2px 6px rgba(0, 0, 0, 0.1);  
}
```

```
.student-list th,  
.student-list td {  
    padding: 12px 15px;  
    text-align: center;  
    border-bottom: 1px solid #eee;  
}
```

```
.student-list th {  
    background-color: #6c5ce7;  
    color: white;  
}
```

```
.student-list tr:hover {  
    background-color: #f1f3f5;  
}
```

```
.student-list button {  
    padding: 6px 12px;  
    background-color: #0984e3;  
    color: white;  
    border: none;  
    border-radius: 6px;  
    cursor: pointer;  
    transition: background-color 0.3s;  
}
```

```
.student-list button:hover {  
    background-color: #74b9ff;  
}
```

```
.student-detail {  
    margin-top: 20px;  
    padding: 20px;  
    background: #f1f3f5;  
    border-radius: 12px;  
}
```



```
.student-detail p {  
  
    margin: 8px 0;  
  
    font-size: 18px;  
  
}
```

```
form {  
  
    width: 500px; /* ★ 폼 너비 늘리기 */  
  
    margin: 40px auto; /* 가운데 정렬 + 위아래 여백 */  
  
    display: flex;  
  
    flex-direction: column;  
  
    align-items: center;  
  
    padding: 80px; /* ★ 폼 안 여백 넓게 */  
  
    padding-left: 40px; /* 가운데 정렬되지 않아서 왼쪽에 padding으로 가운데 정렬함함 */  
  
    border: 1px solid #ddd;  
  
    border-radius: 12px;  
  
    box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1);  
  
    background: #fafafa;  
  
}
```

```
form h2 {  
  
    margin-bottom: 30px;  
  
}
```

```
form div {
```

```
width: 100%;  
  
margin-bottom: 20px;  
}
```

```
form input {  
  
width: 100%;  
  
padding: 14px; /* ★ 입력창 padding도 키워서 여유 있게 */  
  
font-size: 16px;  
  
border: 1px solid #ccc;  
  
border-radius: 8px;  
}
```

```
form button {  
  
margin-top: 20px;  
  
padding: 14px 28px;  
  
font-size: 16px;  
  
background-color: #4CAF50;  
  
color: white;  
  
border: none;  
  
border-radius: 8px;  
  
cursor: pointer;  
  
transition: background-color 0.3s ease;  
}
```

```
form button:hover {  
  
background-color: #45a049;
```

> 14. 다양한 훅 사용법

기본 훅

1. useState

- 설명: 함수형 컴포넌트에서 상태를 관리할 수 있도록 해주는 훅입니다.
- 변수에 값이 변경되면 화면을 다시 그린다.

2. useEffect

useEffect는 React 컴포넌트에서 부작용(side effects)을 처리하는 Hook이에요.

👉 부작용이란? API 호출, 타이머, 이벤트 리스너 등 렌더링과 직접 관련 없는 작업

렌더링이란 화면을 다시 그려주는 작업을 의미한다.

① useEffect 기본 문법

```
useEffect(() => {  
  // 실행할 코드  
  return () => {  
    // 정리(clean-up) 코드 (선택 사항)  
  };  
}, [의존성]);
```

- 렌더링마다 실행 → `useEffect(() => {...})`
 - 처음 한 번만 실행 → `useEffect(() => {...}, [])`
 - 특정 값이 바뀔 때 실행 → `useEffect(() => {...}, [변수])`
 - 타이머, 이벤트 리스너는 정리 필요 → `return () => {...}` 사용
-

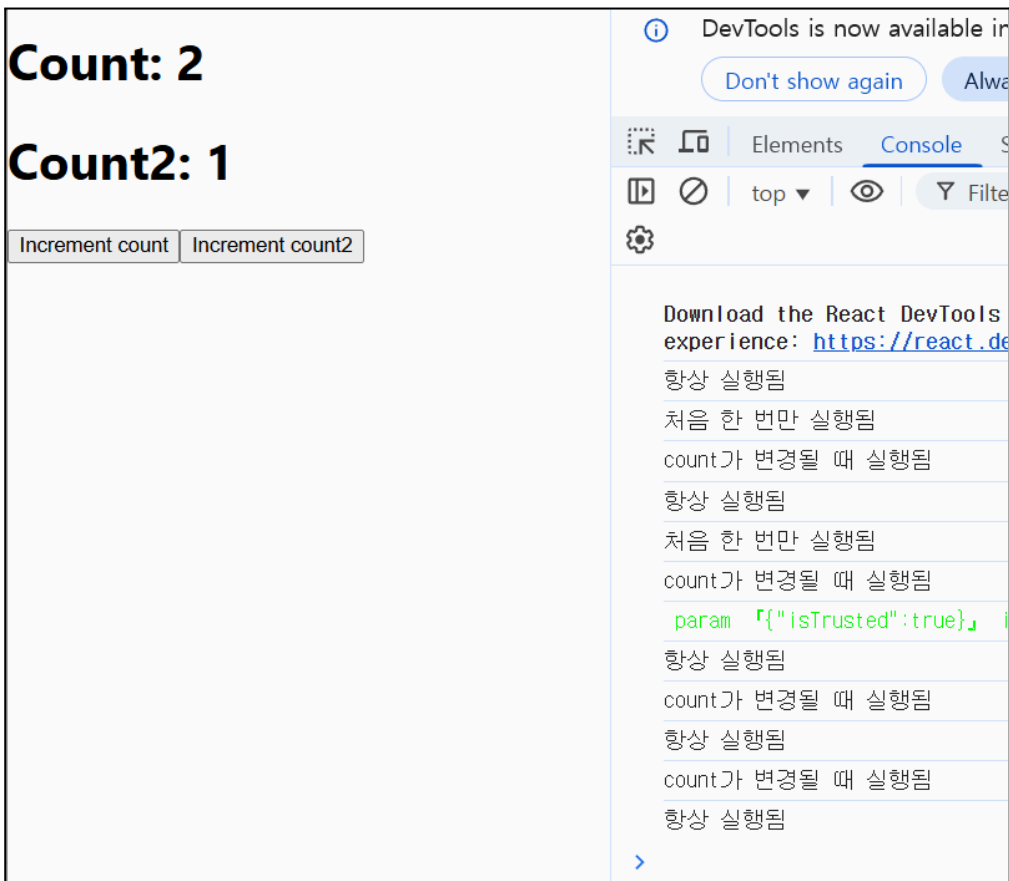
예제 코드

```
useEffect(() => {  
  console.log('항상 실행됨');  
});
```

```
useEffect(() => {  
  console.log('처음 한 번만 실행됨');  
}, []);
```

```
useEffect(() => {  
  console.log('count가 변경될 때 실행됨');  
}, [count]);
```

`useEffect` 혹은 테스트하기 위한 전체 `React` 컴포넌트를 아래에 작성해 드릴게요. 이 예제에서는 `count` 상태를 증가시키는 버튼을 통해 `useEffect` 동작을 확인할 수 있습니다.



The screenshot displays a web application interface on the left and the React DevTools console on the right. The web app shows two counts: 'Count: 2' and 'Count2: 1'. Below these are two buttons labeled 'Increment count' and 'Increment count2'. The DevTools console on the right shows a series of log messages: '항상 실행됨' (always runs), '처음 한 번만 실행됨' (runs once), and 'count가 변경될 때 실행됨' (runs when count changes), demonstrating the behavior of different `useEffect` configurations.

```
import React, { useState, useEffect } from 'react';

function EffectExample() {
  const [count, setCount] = useState(0);
  const [count2, setCount2] = useState(0);

  // 항상 실행됨 (렌더링될 때마다)
  useEffect(() => {
    console.log('항상 실행됨');
  });

  // 처음 한 번만 실행됨 (마운트 시)
  useEffect(() => {
    console.log('처음 한 번만 실행됨');
  }, []);

  // count가 변경될 때 실행됨
  useEffect(() => {
    console.log('count가 변경될 때 실행됨');
  }, [count]);

  return (
    <div>
      <h1>Count: {count}</h1>
      <h1>Count2: {count2}</h1>
      <button onClick={() => setCount(prev => prev + 1)}>Increment count</button>
      <button onClick={() => setCount2(prev => prev + 1)}>Increment count2</button>
    </div>
  );
}

export default EffectExample;
```

3 useEffect의 클린업 (정리 함수)

타이머, 이벤트 리스너 같은 것들은 꼭 정리해줘야 함!

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

function TimerComponent() {
  useEffect(() => {
    const interval = setInterval(() => {
      console.log('1초마다 실행됨');
    }, 1000);

    return () => {
      clearInterval(interval);
      console.log('타이머 제거됨');
    };
  }, []);

  return <div>타이머가 실행 중입니다. 콘솔을 확인하세요.</div>;
}

export default TimerComponent;
```

동작 설명

1. 컴포넌트가 처음 마운트될 때 ([]) 의존성 배열 덕분에
 - `setInterval`이 실행되어 1초마다 `console.log('1초마다 실행됨')`을 출력합니다.
2. 컴포넌트가 언마운트될 때

- `return`으로 정의한 함수가 실행됩니다.
- `clearInterval(interval)`이 실행되어 1초마다 실행되던 타이머를 제거합니다.
- `console.log('타이머 제거됨')`이 출력됩니다.

왜 정리가 필요할까?

- 만약 `clearInterval`을 하지 않으면, 컴포넌트가 사라져도 계속 타이머가 돌아가면서 메모리 누수나 불필요한 작업이 발생할 수 있어요.
 - 그래서 React는 `useEffect` 안에서 리소스를 사용하는 경우, 정리(**clean-up**) 함수를 `return`으로 지정할 수 있도록 해줍니다.
-

④ 실제 사용 예제 (API 호출)

```
useEffect(() => {  
  fetch('https://jsonplaceholder.typicode.com/posts')  
    .then(res => res.json())  
    .then(data => console.log(data));  
}, []); // 처음 한 번만 실행
```


title: Hello World! (11)

Button clicked: 11 times

Change Title

Title이 변경될 때마다 실행되도록 구현한 `useEffect`예제 입니다.

App.js로 만들기

```
import React, { useState, useEffect } from 'react';

// TitleChanger 컴포넌트
const TitleChanger = () => {
  const [title, setTitle] = useState('Hello World!');
  const [clickCount, setClickCount] = useState(0);

  useEffect(() => {
    document.title = title; // 브라우저의 탭 제목을 변경합니다.
    console.log('컴포넌트가 렌더링되었습니다!');
    return () => {
      console.log('컴포넌트가 제거됩니다.');
    };
  }, [title]);

  const handleClick = () => {
    setClickCount(clickCount + 1);
    setTitle(`Hello World! (${clickCount + 1})`); // 클릭 수에 따라 제목 변경
  };

  return (
    <div>
      <p>title: {title}</p>
      <p>Button clicked: {clickCount} times</p>
      <button onClick={handleClick}>Change Title</button>
    </div>
  );
};
```

```
// App 컴포넌트
const App = () => {
  return (
    <div>
      <h1>Welcome to Title Changer App!</h1>
      <TitleChanger /> {/* TitleChanger 컴포넌트 포함 */}
    </div>
  );
};

export default App;
```

json 요청하기

<https://jsonplaceholder.typicode.com/users> 에 들어가 보면 test json 데이터를 확인할 수 있다.

```
import React, { useState, useEffect } from 'react';

// DataFetcher 컴포넌트 정의
const DataFetcher = () => {
  const [data, setData] = useState(null);

  useEffect(() => {
    // 데이터 fetch
    fetch('https://jsonplaceholder.typicode.com/users')
      .then(response => response.json())
      .then(data => setData(data))
      .catch(error => console.error('Error fetching data:', error));
  }, []); // 빈 배열을 의존성 배열로 전달하면, 컴포넌트가 처음 렌더링될 때만 실행됩니다.
```

```
  return (
    <div>
      {data ? (
        <div>
          {data.map(user => (
            <p key={user.id}>{user.name}</p>
          ))}
        </div>
      ) : (
        <p>Loading...</p>
      )}
    </div>
  );
};
```

```
// App 컴포넌트 정의
const App = () => {
  return (
    <div style={{ padding: '20px' }}>
      <h1>User List</h1>
      <DataFetcher />
    </div>
  );
};
```

```
export default App;
```

설명

1. **data** 배열의 사용:

- **data**는 여러 사용자 객체로 이루어진 배열입니다. 따라서 **data.name**이 아닌 **data.map(user => user.name)**을 사용해야 합니다.

2. **map** 메소드:

- **data.map(user => ...)**을 사용하여 배열의 각 사용자 객체에 대해 **<p>** 요소를 생성합니다.
- **key={user.id}**를 추가하여 각 **<p>** 요소에 고유한 키를 부여합니다. 이는 **React**가 각 요소를 효율적으로 추적하는 데 필요합니다.

3. 에러 처리:

- **.catch(error => console.error('Error fetching data:', error))**를 추가하여 데이터를 가져오는 동안 발생할 수 있는 오류를 콘솔에 출력합니다.

이 수정된 코드로 **data** 배열의 각 사용자 이름을 웹 페이지에 렌더링할 수 있으며, 데이터가 로드되는 동안 **"Loading..."** 메시지를 표시합니다.

fetch관련 자바 스크립트 예제

`fetch`는 웹 브라우저에서 HTTP 요청을 보내고 응답을 받기 위한 API입니다. `fetch`는 JavaScript에서 비동기적으로 데이터를 가져오는 데 사용되며, 주로 API 호출, 데이터 요청, 서버와의 상호작용 등에 사용됩니다.

기본 문법

```
fetch(url, options)
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.json(); // 또는 response.text() 등
  })
  .then(data => {
    console.log(data);
  })
  .catch(error => {
    console.error('There has been a problem with your fetch operation:',
error);
  });
```

주요 개념

1. **URL**: 요청을 보낼 서버의 URL입니다.
2. 옵션 객체: HTTP 메서드(GET, POST, PUT, DELETE 등), 헤더, 본문 데이터 등을 설정할 수 있습니다.
3. 응답 객체: 서버에서 반환된 응답을 나타내는 객체입니다. `response.json()`은 JSON 형태로 데이터를 파싱합니다.

사용 예제

1. GET 요청

서버로부터 데이터를 가져오는 기본적인 GET 요청 예제입니다.

```
fetch('https://api.example.com/data')
```

```
.then(response => {
  if (!response.ok) {
    throw new Error('Network response was not ok');
  }
  return response.json();
})
.then(data => {
  console.log('Data:', data);
})
.catch(error => {
  console.error('Fetch error:', error);
});
```

- 설명: **fetch**를 호출하여 서버에서 데이터를 요청합니다. 응답을 **JSON**으로 변환하고, 데이터를 콘솔에 출력합니다. 오류가 발생하면 **catch** 블록에서 처리합니다.

2. POST 요청

서버에 데이터를 전송하는 **POST** 요청 예제입니다.

```
fetch('https://api.example.com/data', {
  method: 'POST', // 요청 메서드
  headers: {
    'Content-Type': 'application/json' // 요청 헤더
  },
  body: JSON.stringify({ key: 'value' }) // 요청 본문
})
.then(response => {
  if (!response.ok) {
    throw new Error('Network response was not ok');
  }
  return response.json();
})
.then(data => {
  console.log('Response Data:', data);
});
```

```
  })  
  .catch(error => {  
    console.error('Fetch error:', error);  
  });
```

- 설명: `method`를 'POST'로 설정하고, `body`에 JSON 문자열을 포함시킵니다. 이 데이터를 서버에 전송하고, 응답을 처리합니다.

3. Error Handling

`fetch` 호출 시 발생할 수 있는 오류를 처리하는 예제입니다.

javascript

코드 복사

```
fetch('https://api.example.com/data')  
  .then(response => {  
    if (!response.ok) {  
      throw new Error('Network response was not ok');  
    }  
    return response.json();  
  })  
  .then(data => {  
    console.log('Data:', data);  
  })  
  .catch(error => {  
    console.error('Fetch error:', error.message);  
  });
```

- 설명: `response.ok`를 사용하여 응답 상태가 성공적인지 확인하고, 오류가 발생하면 `catch` 블록에서 처리합니다.

비동기/대기 (`async/await`)

`fetch`와 `async/await`를 함께 사용하여 코드의 가독성을 높일 수 있습니다.

javascript

코드 복사

```
const fetchData = async () => {
```



```
try {
  const response = await fetch('https://api.example.com/data');
  if (!response.ok) {
    throw new Error('Network response was not ok');
  }
  const data = await response.json();
  console.log('Data:', data);
} catch (error) {
  console.error('Fetch error:', error.message);
}
};

fetchData();
```

- 설명: `fetch`를 `await`하여 응답을 기다리고, `JSON`으로 변환 후 데이터를 처리합니다. 오류가 발생하면 `catch` 블록에서 처리합니다.

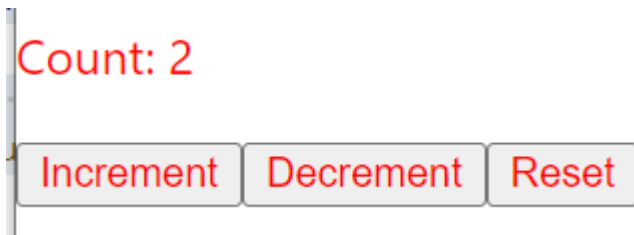
결론

`fetch`는 웹에서 데이터를 가져오거나 서버에 데이터를 보내는 데 유용한 **API**입니다. 비동기적으로 동작하며, `Promise` 기반의 처리를 지원하여 코드의 가독성을 높입니다. `async/await`와 함께 사용하면 더욱 깔끔한 코드를 작성할 수 있습니다.

useReducer

- 설명: `useState`보다 복잡한 상태 로직을 관리할 때 사용되는 훅입니다. 상태 업데이트 로직을 별도의 `reducer` 함수로 분리할 수 있습니다.

`useReducer`는 상태 관리 로직을 리듀서 함수로 분리하여 복잡한 상태 업데이트를 간단하고 예측 가능하게 처리할 수 있는 `React` 훅입니다.



다음과 같은 화면은 3개의 메소드로 `count`값을 조작하는 형태로 되어 있다.

이렇게 여러개의 데이터와 여러개의 메소드가 같은 데이터를 조작하는 용도로 사용되면 `useReducer`로 묶어서 표현할 수 있다.

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

// 1. 리듀서 함수 정의
const reducer = (state, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return { count: state.count + 1 };
    case 'DECREMENT':
      return { count: state.count - 1 };
    case 'RESET':
      return { count: 0 };
    default:
      return state;
  }
};

// 2. 초기 상태
const initialState = { count: 0 };
const Counter = () => {
  // 3. useReducer를 사용하여 상태와 디스패치 함수를 정의
  const [state, dispatch] = useReducer(reducer, initialState);
```

```

return (
  <div>
    {/* 4. 변경된 데이터를 그때그때 보여줄 state변수 기술*/}
    <p>Count: {state.count}</p>
    {/* 5. 디스패치 함수를 사용하여 액션을 보냄 */}
    <button onClick={() => dispatch({ type: 'INCREMENT' })}>Increment</button>
    <button onClick={() => dispatch({ type: 'DECREMENT' })}>Decrement</button>
    <button onClick={() => dispatch({ type: 'RESET' })}>Reset</button>
  </div>
);
};

export default Counter;

```

`const [state, dispatch] = useReducer(reducer, initialState);` 에서
`state`는 현재 상태를 나타내는 변수입니다.
`dispatch`는 상태를 업데이트하기 위해 액션을 보내는 함수입니다

1. `useReducer`에서 사용할 로직들을 구현한다.
2. 값변경시 화면에 변경된 값을 바로바로 표현해줄 변수를 생성한다.
3. 로직과 변수로 `useReducer`메소드를 이용해서 `state`와`dispatch`를 얻는다.
4. 4. 변경된 데이터를 그때그때 보여줄 `state`변수 기술
5. 디스패치 함수를 사용하여 액션을 보냄

사용자 정의 훅 (Custom Hook)

사용자 정의 훅을 통해 공통된 로직을 재사용할 수 있습니다.

많이 사용하는 기능을 Hook으로 만들어서 필요할때 사용한다. 다음예제는 토글 기능을 가지고 있는 `useToggle` Hook를 만들어 보았다.

`useToggle`은 불리언 상태를 간단하게 토글할 수 있는 사용자 정의 훅입니다.

주요 포인트:

- 목적: 불리언 값을 `true`와 `false`로 쉽게 변경하는 기능을 제공합니다.
- 구성: 상태 값과 상태를 반전시키는 함수(토글 함수)를 반환합니다.
- 장점:
 - 코드 간결성: 상태를 간단하게 관리할 수 있음.
 - 재사용성: 여러 컴포넌트에서 반복 사용 가능.
 - 초기 값 설정: 초기 상태를 `true` 또는 `false`로 설정 가능.

App Component

Off

Toggle1

Off

Toggle2

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

// 사용자 정의 훅: true/false 값을 토글

const useToggle = (initialValue = false) => {

  const [value, setValue] = useState(initialValue);

  const toggle = () => {
```

```

    setValue(prevValue => !prevValue);

};

return [value, toggle];

};

// Toggle 버튼을 포함한 컴포넌트

const ToggleComponent = () => {

    const [isToggled1, toggle1] = useToggle(); // 사용자 정의 훅 사용
    const [isToggled2, toggle2] = useToggle(); // 사용자 정의 훅 사용

    return (<>

        <div>

            <h1>{isToggled1 ? 'On' : 'Off'}</h1>

            <button onClick={toggle1}>Toggle1</button>

        </div>

        <div>

            <h1>{isToggled2 ? 'On' : 'Off'}</h1>

            <button onClick={toggle2}>Toggle2</button>

        </div>

    </>);

};

// App 컴포넌트

```

```
const App = () => {  
  
  return (  
  
    <div>  
  
      <h1>App Component</h1>  
  
      <ToggleComponent /> { /* ToggleComponent를 포함 */}  
  
    </div>  
  
  );  
};  
  
export default App;
```

> 20.

> 21.

> 22.

> 23.

> 24.

> 25.

> 26.

> 27.

> 28.

> 29.

> 30.
