



# 애프로그래밍 3주차 2차시 (실습)

주제: React 계산기 앱 만들기



담당교수 안효정 | 인공지능학과



## 학습 목표

3주차 2차시 실습을 통해 배울 내용

- ▶ React 이벤트 처리 심화
- ▶ State 를 활용한 입력값/결과 관리
- ▶ 조건부 렌더링 으로 오류 처리
- ▶ 리스트 렌더링 으로 계산 기록 관리

👉 하나의 작은 프로젝트(계산기 앱) 속에서 React 기본기를 종합적으로 익히는 것이 목표



# 오늘의 실습 로드맵

React 계산기 앱 만들기 과정

- 1 프로젝트 준비
- 2 계산기 UI 설계와 컴포넌트 분리
- 3 버튼 입력과 State 관리
- 4 연산 처리(=, C)
- 5 조건부 렌더링으로 오류 처리
- 6 계산 기록 기능
- 7 Keypad 그룹화 & CSS 리팩토링
- 8 학습 정리 & 과제 안내

👉 단계별 실습을 통해 React의 핵심 개념을 종합적으로 활용해 봅니다

# Part A



## 프로젝트 준비

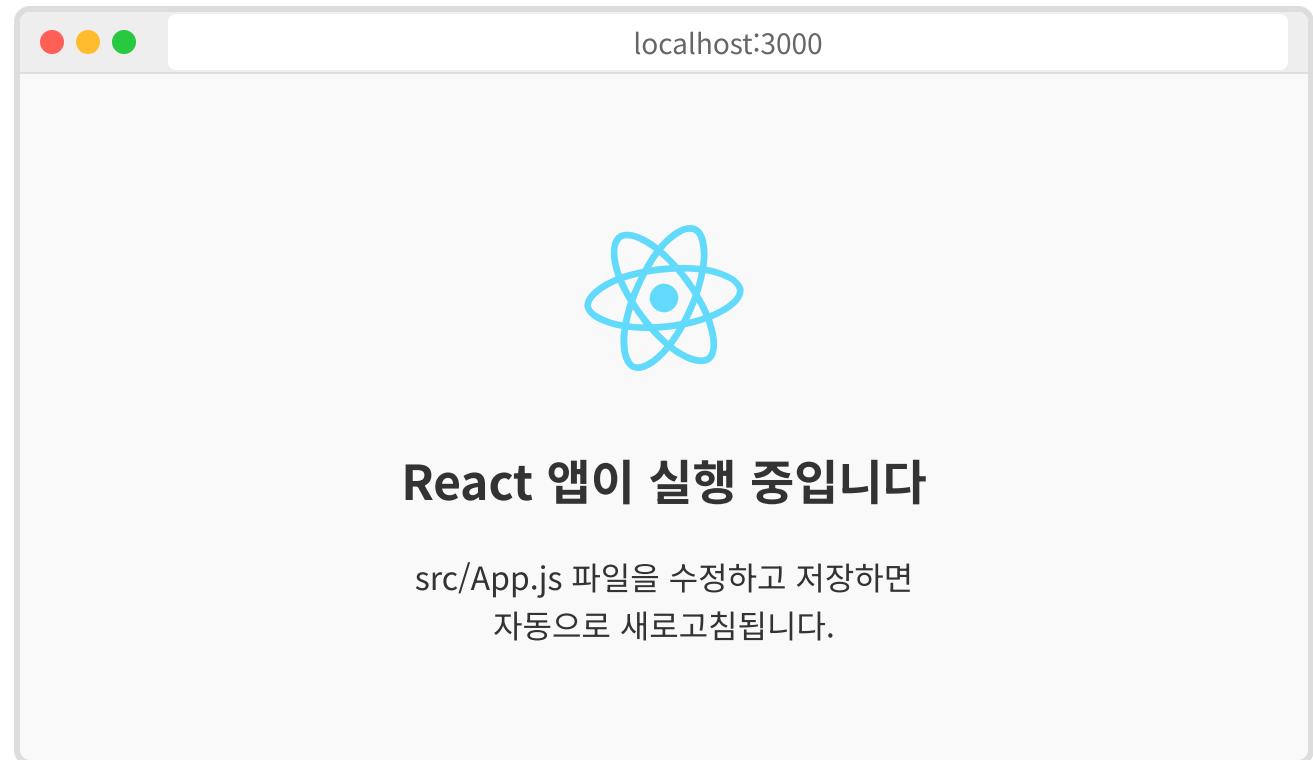
---

# 프로젝트 생성 (create-react-app)

```
npx create-react-app  
calculator-app  
cd calculator-app  
npm start
```



브라우저에서 React 기본 화면  
(localhost:3000)이 뜨면 성공



# 폴더 구조 정리

```
src/  
components/  
Button.jsx  
Display.jsx  
History.jsx  
Keypad.jsx  
App.js  
App.css
```

## 각 파일의 역할

App.js	메인 컴포넌트, 상태 관리 및 하위 컴포넌트 조합
Button.jsx	숫자/연산자 버튼 UI 및 클릭 이벤트 처리
Display.jsx	계산기 화면 표시 담당 (입력값, 결과값)
History.jsx	이전 계산 기록 목록 렌더링
Keypad.jsx	버튼 배열을 그룹화하여 레이아웃 구성
App.css	전체 앱 스타일 정의

💡 컴포넌트 분리로 코드 재사용성 향상 및 유지보수 용이

💡 기능별 컴포넌트를 components 폴더에 분리  
하여 관리

# App.js 초기화

src/App.js

```
// 함수형 컴포넌트 정의
function App() {
  return (
    <div className="container">
      <h1>React 계산기</h1>
    </div>
  );
}

export default App;
```

## JSX 기본 문법

- ✓ **HTML과 유사하지만 다름:** className, htmlFor 등 카멜케이스 속성명 사용
- ✓ **반환값은 하나의 요소로 감싸야 함:** 형제 요소는 하나의 부모로 묶어야 함
- ✓ **자바스크립트 표현식 사용:** 중괄호 {expression}로 JS 코드 삽입 가능
- ✓ **함수형 컴포넌트:** 함수가 JSX 반환하는 간결한 구조
- ✓ **export default:** 다른 파일에서 import하여 사용 가능

💡 브라우저에서 제목이 잘 보이는지 확인하세요

# Part B



## UI 설계/컴포넌트화



# 계산기 UI 요소 도출

## Display(결과)

사용자 입력과 계산 결과를 표시

## 숫자 버튼(0~9)

숫자 입력을 위한 키패드

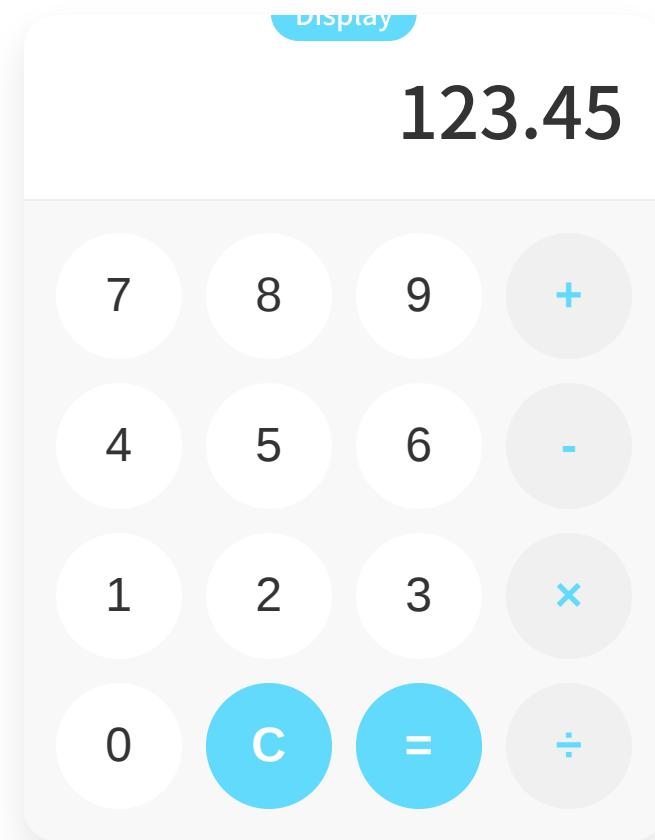
## 연산자 버튼(+, -, ×, ÷)

사칙연산을 위한 기능 버튼

## 특수 버튼(=, C)

계산 실행과 초기화 기능

💡 이들을 컴포넌트로 나누어 작성



# Display 컴포넌트 작성

```
//  
src/components/Display.jsx  
export default function  
Display({ value }) {  
return (  


{value}  
</div>  
);  
}


```

💡 props로 전달된 value를 표시하는 간단한 컴포넌트

App 컴포넌트  
const [input, setInput] =  
useState("123");

Display 컴포넌트  
<div className="display">123</div>

value="123"

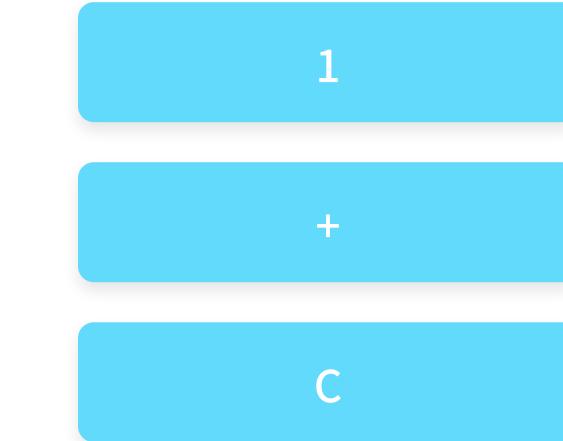
props는 부모 컴포넌트(App)에서 자식 컴포넌트(Display)로 데이터를 전달하는 방식입니다.

1. **비구조화 할당**으로 props 객체에서 value를 바로 추출
2. JSX 내부에서 **중괄호 {}**를 사용해 JavaScript 표현식으로 value 값을 출력
3. 단방향 데이터 흐름: 부모→자식으로만 데이터가 전달됨

# Button 컴포넌트 작성

```
// src/components/Button.jsx  
  
export default function Button({  
  label, onClick }) {  
  
  return (  
    <button className="btn" onClick=  
      {() => onClick(label)}>  
    {label}  
  </button>  
);  
}
```

- 💡 공통 버튼 컴포넌트로 설계하여 **label**과 **onClick**만 달라지게 함으로써 코드 중복을 최소화하고 일관된 UI를 유지합니다.



## Props 활용 방식

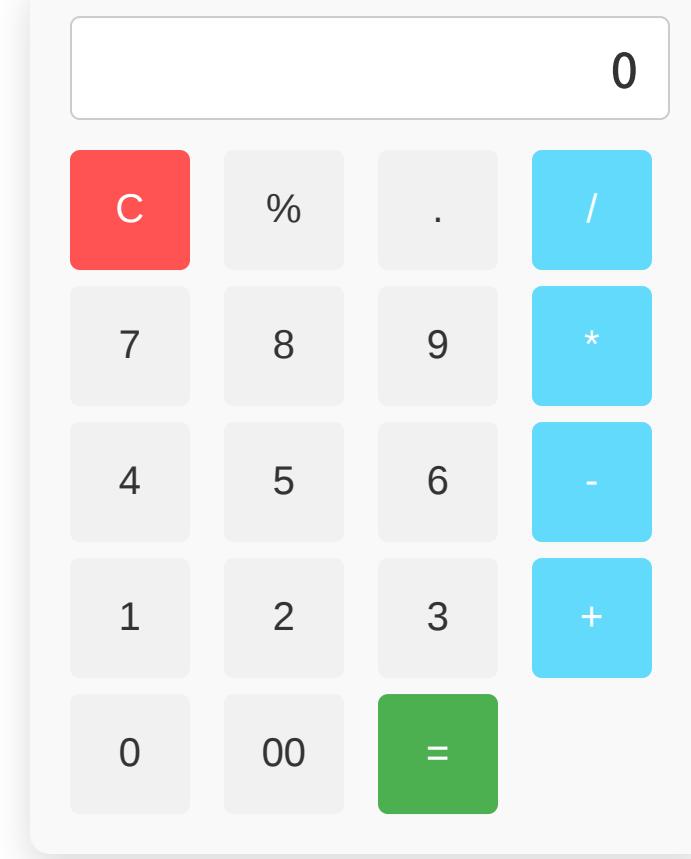
<b>label</b>	버튼에 표시될 텍스트 또는 기호 (예: 숫자, 연산자)
<b>onClick</b>	버튼 클릭 시 실행될 함수 (클릭한 label 값을 상위 컴포넌트로 전달)
→ 버튼 디자인과 동작을 한 번만 정의하고, 다양한 용도로 재사용 가능	

## 기본 스타일(App.css)

```
/* 계산기 컨테이너 스타일 */
.container {
  max-width: 300px;
  margin: 0 auto;
  text-align: center;
}

/* 결과 화면 스타일 */
.display {
  height: 50px;
  border: 1px solid #ccc;
  margin-bottom: 8px;
  font-size: 24px;
}

/* 버튼 공통 스타일 */
.btn {
  width: 60px;
  height: 60px;
  margin: 4px;
  font-size: 20px;
}
```



**container**: 전체 계산기 영역 중앙 정렬

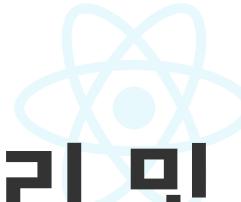
**display**: 계산 결과를 표시하는 영역

**btn**: 모든 버튼에 적용되는 공통 스타일

👉 이후에 그리드 레이아웃으로 개선 예정

💡 단순한 UI지만 가독성을 높이는 구조

# Part C



## State 관리 및 입력 처리

---

# App.js에 State 추가하기

```
// useState Hook 불러오기
import { useState } from
"react";
import Display from
"./components/Display";
import Button from
"./components/Button";
function App() {
// 입력값을 위한 상태 변수
const [input, setInput] =
useState("");
const handleClick =
(value) => {
setInput(prev => prev +
value);
};
return (
/* JSX 반환 */
);
}
```

## useState 기본 구조

- ✓ `const [state, setState] = useState(initialValue)` - 배열 구조 분해를 통해 상태값과 업데이트 함수를 얻음
- ✓ React는 컴포넌트 렌더링 간 상태를 기억하고 유지함

이전 상태

"123"

setState 호출

prev + "+"

새 상태

"123+"

## useState 모범 사례

- ✓ 이전 상태 기반 업데이트 시 함수형 업데이트 사용: `setInput(prev => prev + value)`
- ✓ 객체/배열은 항상 새 참조로 업데이트 (불변성 유지)
- ✓ 초기화 비용이 크다면 초기화 함수 사용: `useState(() => 복잡한_초기화())`

💡 클릭 이벤트 발생 시 입력값이 화면에 누적되어 표시됨

# 이벤트 처리: handleClick

## 함수 예시

```
// 버튼 클릭 이벤트 처리 함수  
const handleClick = (value)  
=> {  
  setInput(prev => prev +  
  value);  
};  
  
// 버튼 컴포넌트에 이벤트 핸들러  
전달  
  
<Button  
  label="1"  
  onClick={handleClick}  
/>
```

💡 클릭 이벤트 발생 시 label 값이 handleClick 으로 전달되어 input 상태 업데이트



### 버튼 클릭

사용자가 버튼 요소를 클릭



### 합성 이벤트(Synthetic Event)

React가 브라우저 네이티브 이벤트를 합성 이벤트로 변환



### 핸들러 실행

handleClick 함수가 실행되어 상태 업데이트



### 리렌더링

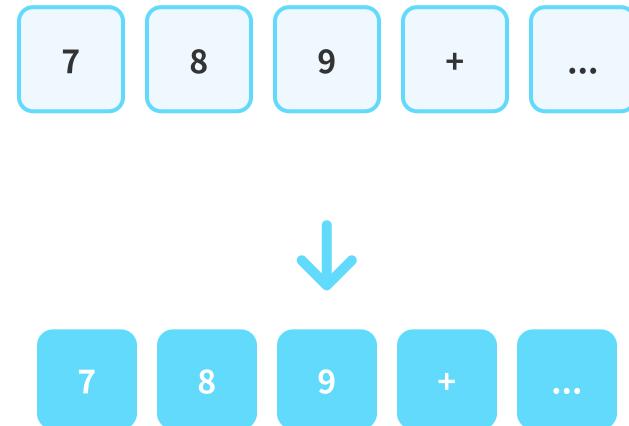
상태 변경으로 컴포넌트 재렌더링

## React 이벤트 처리 모범 사례

- ✓ 이벤트 핸들러 이름은 **handle**로 시작
- ✓ 직접 호출 `onClick={handleClick()}`가 아닌 함수 참조 `onClick={handleClick}` 사용
- ✓ 상태 업데이트 시 함수형 업데이트 `prev => prev + value` 활용

# 버튼 배열 출력(map 활용)

```
const buttons =  
["7", "8", "9", "+", "4", "5", "6", "-  
", "1", "2", "3", "*", "0", "C", "=", "/"];  
  
{buttons.map((b, idx) => (  
  
<Button  
key={idx}  
label={b}  
onClick={handleClick}  
/>  
)})
```



## ⚠️ key 속성의 중요성

리액트는 key 속성을 통해 어떤 항목이 변경, 추가 또는 제거되었는지 식별합니다. 배열의 각 요소에는 고유한 key가 있어야 합니다.

실제 프로덕션 환경에서는 인덱스보다 고유 ID를 사용하는 것이 권장됩니다. 인덱스는 항목이 재정렬되면 변경될 수 있기 때문입니다.

- 💡 map() 메서드를 사용하면 배열의 각 요소를 반복하여 컴포넌트로 변환할 수 있습니다. 유지보수가 쉽고 코드가 간결해집니다.

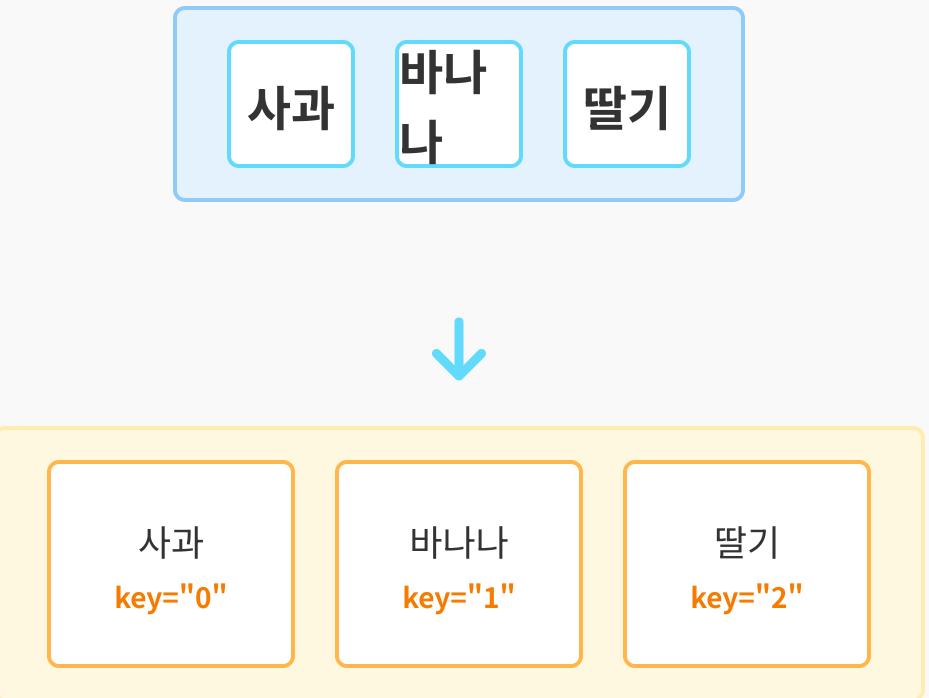
# 리스트 렌더링 원리

```
// 배열 데이터  
const items = ["사과", "바나나", "딸  
기"];  
  
// map으로 JSX 변환  
return (  


- {items.map((item, index) => (  
<li key={index}>{item}</li>  
))}  
</ul>  
);

```

## 리스트 렌더링 작동 원리



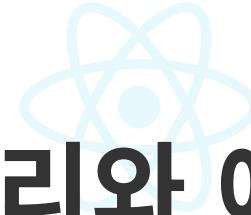
## key 속성이 필수인 이유

- React가 변경된 항목을 효율적으로 파악
- 불필요한 리렌더링 방지
- DOM 조작 최소화로 성능 향상
- 고유 식별자 사용 권장 (index는 비권장)

배열 데이터 → map 함수 → React 컴포넌트 변환

계산기 앱에서는 버튼과 기록에 리스트 렌더링 활용

# Part D



## 연산 처리와 예외처리

---

## = 버튼 처리 (계산)

```
if (value === "=") {  
try {  
setInput(eval(input).toString());  
} catch {  
setInput("Error");  
}  
  
return;  
}
```

### ⚠ 보안 경고: eval() 사용 주의

eval() 함수는 **보안 취약점을** 발생시킬 수 있습니다:

- 입력된 문자열을 JavaScript 코드로 직접 실행
- 악의적인 코드 실행 가능성 (XSS, 코드 인젝션)
- 실제 서비스에서는 절대 사용 금지

ⓘ 학습 목적으로 실제 프로젝트에서는 math.js, expr-eval 등 안만 사용합니다. 전한 수식 파서 라이브러리를 사용해야 합니다.

💡 try-catch로 잘못된 수식 오류 처리

# C 버튼(초기화) 처리

```
if (value === "C") {  
  setInput("");  
  return;  
}
```



C 버튼은 입력값을 초기화하여 계산기를 리셋합니다.

7	8	9	/
4	5	6	*
1	2	3	-
C	0	.	+



## State 초기화 특징:

- useState 상태값을 빈 문자열("")로 변경
- setState 함수 호출 시 리렌더링 발생
- return으로 함수 실행 종료 (다른 코드 실행 방지)

7	8	9	/
4	5	6	*
1	2	3	-
C	0	.	+

# eval 사용의 보안 이슈

## ☒ 코드 인젝션(Code Injection) 위험

사용자 입력값이 그대로 코드로 실행되어 악성 코드가 시스템에서 동작할 수 있음

## ☒ XSS(Cross-Site Scripting) 취약점

eval()을 통해 입력된 악성 스크립트가 실행되어 사용자 데이터 탈취 가능

## ☒ 전역 스코프 오염

평가된 코드가 현재 스코프에 접근하여 변수와 함수를 수정할 수 있음

### ⚠️ 실무 경고

계산기 앱에서 eval() 사용은 학습 목적으로만 허용됩니다. 실제 서비스 개발에서는 절대 사용하지 마세요!



악성 입력

"2+2; alert('해킹 됨!')"



eval() 실행

eval("2+2; alert('해킹됨!')")



보안 취약점 악용

악성 코드 실행 및 정보 탈취

위험

```
// 사용자 입력값을 직접 eval로 실행
function calculate(input) {
  return eval(input); // ❌ 보안 취약점!
}
```

안전

```
// math.js 라이브러리 사용
import { evaluate } from 'mathjs';
function calculate(input) {
  return evaluate(input); // ✅ 안전한 수학식 평가
}
```

# 수식 파서(math.js, expr-eval 등)

## math.js

npm i mathjs

확장 가능한 수학 라이브러리

- 복잡한 연산, 단위 변환 지원
- 다양한 수학 함수 내장
- 타입 체크 및 오류 검증

## expr-eval

npm i expr-eval

경량화된 수식 평가 도구

- 간결하고 빠른 성능
- 기본 연산자 및 함수 제공
- 안전한 입력 검증

💡 실무에서는 `eval()` 대신 **안전한 수식 파서**를 사용하세요! 보안, 성능, 기능성 모두 향상됩니다.

```
// math.js 사용 예시
import { evaluate } from 'mathjs';
// 사용자 입력 문자열 계산
const result = evaluate('2 * (3 + 4)');
console.log(result); // 14
// 고급 수학 기능 사용
evaluate('sin(45 deg) * sqrt(16)'); // 2.83
```

```
// expr-eval 사용 예시
import { Parser } from 'expr-eval';
const parser = new Parser();
const expr = parser.parse('2 * (3 + 4)');
const result = expr.evaluate(); // 14
// 변수 활용
expr.evaluate({ x: 5 }); // x 변수 사용 가능
```

💡 계산기 앱 완성 후 `eval() → 수식 파서`로 교체하는 실습도 시도해보세요!

# eval vs 수식 파서 요약 비교

비교 항목	eval()	수식 파서 (math.js, expr-eval)
보안성	<span style="color: red;">✖️</span> 매우 위험 XSS, 코드 인젝션 취약점	<span style="color: green;">✓</span> 매우 안전 수학적 연산만 제한적 허용
안전성	<span style="color: orange;">⚠️</span> 위험함 임의 코드 실행 가능	<span style="color: green;">✓</span> 안전함 샌드박스 환경에서 제한적 실행
실무 적용성	<span style="color: red;">✖️</span> 실무에서 사용 금지 보안 감사에서 취약점으로 분류	<span style="color: green;">✓</span> 실무 표준 안전하고 확장 가능한 기능 제공
사용 사례	<span style="color: orange;">⚠️</span> 학습용, 데모용 신뢰할 수 있는 환경에서만	<span style="color: green;">✓</span> 실제 서비스, 상용 앱 고객 대면 서비스에 안전하게 사용

**권장사항:** 실제 프로젝트에서는 [math.js](#) 또는 [expr-eval](#) 같은 수식 파서 라이브러리를 사용하세요. eval()은 학습 목적으로만 제한적으로 사용하는 것이 좋습니다.

i 현업에서는 코드 검사 도구(ESLint)가 eval() 사용을 금지하는 룰을 기본적으로 포함하고 있습니다.

# 수식 파서 개념 및 동작 원리

## 파서(Parser)란?

문자열로 작성된 수식을 구조화된 형태로 해석하여 계산하는 프로그램 또는 로직입니다.

## 수식 파서의 장점

- 수학적 연산자 우선순위 준수
- 보안 위험 없이 안전하게 수식 평가
- 구조화된 결과를 다양한 방식으로 활용 가능

math.js, expr-eval 같은 라이브러리가 이러한 원리로 동작

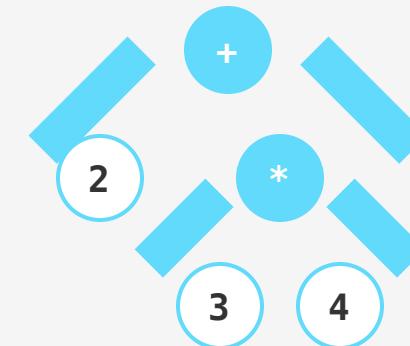
### 1 입력 문자열

"2 + 3 \* 4"

### 2 토큰 분해

2 + 3 \* 4

### 3 트리 구성



연산자 우선순위: 곱셈(\*)이 덧셈(+)보다 우선

### 4 계산 실행

$$2 + (3 * 4) = 14$$

# handleClick 함수 완성본

```
const handleClick = (value) => {
  // C 버튼 처리 (초기화)
  if (value === "C") {
    setInput("");
    return;
  }
  // = 버튼 처리 (계산 실행)
  if (value === "=") {
    try {
      // 학습용으로만 eval 사용 (실무에서는 권장하지 않음)
      const result = eval(input).toString();
      setInput(result);
      // 기록에 추가
      setHistory(prev => [input + " = " + result,
        ...prev]);
    } catch {
      setInput("Error");
    }
    return;
  }
  // 일반 입력 처리 (숫자, 연산자 등)
  setInput(prev => prev + value);
};
```

## C 버튼 처리

입력값을 초기화하고 함수를 즉시 종료합니다.  
setInput("")으로 빈 문자열 설정 후 return으로 함수 종료

## = 버튼 처리

입력된 수식을 계산하고 결과를 표시합니다.  
try-catch로 오류를 처리하고, 결과를 기록에 추가합니다.

## 일반 입력 처리

이전 입력값(prev)에 새 값을 추가합니다.  
함수형 업데이트로 최신 상태를 보장합니다.

# 조건부 렌더링 - 오류 메시지

```
// 조건부 렌더링 - && 연산자 활용
{input === "Error" && ( <p
style={{color:"red"}}> 잘못된 수식! </p> )}
```

```
// 조건부 렌더링 - 삼항 연산자 활용
{input === "Error" ? <p style=
{{color:"red"}}>잘못된 수식!</p>
: null}
```

- React에서 조건부 렌더링은 특정 조건이 참일 때만 UI 요소를 표시하는 기법입니다. **&&** 연산자나 **삼항 연산자**를 주로 사용합니다.

## 일반 상태 (input: "2+3")

2+3

## ▲ 오류 상태 (input: "Error")

Error

● 잘못된 수식!

1

상태 확인  
(input === "Error"?)

2

조건 충족 시  
오류 메시지 요소 생성

3

DOM에 오류  
메시지 렌더링

# Part E



## 계산 기록 기능

---

# History 컴포넌트 생성

```
//  
src/components/History.jsx  
  
export default function  
History({ records }) {  
  
return (  
  
<ul>  
  
{records.map((r, idx) => (  
  
<li key={idx}>{r}</li>  
))}  
  
</ul>  
);  
}
```

## 계산 기록

2 + 3 = 5

10 - 4 = 6

7 \* 8 = 56

25 / 5 = 5

💡 App.js에서 history 배열을 관리하고 props로 전달

- ✓ records props를 배열로 받아 리스트 렌더링
- ✓ map 함수로 각 기록을 li 요소로 변환
- ✓ key={idx}로 각 항목에 고유 식별자 부여

# 기록 State 관리

```
// 계산 기록을 저장할 state 추가  
import { useState } from "react";  
  
function Calculator() {  
  const [input, setInput] =  
    useState("");  
  
  const [history, setHistory] =  
    useState([]);  
  
  // 계산 후 기록 추가하는 로직  
  const handleEqual = () => {  
    try {  
      const result = eval(input);  
  
      const newRecord = `${input} = ${result}`;  
  
      // 최신 기록이 맨 위로 오도록  
      setHistory([newRecord,  
      ...history]);  
  
      setInput(String(result));  
    } catch (error) {  
      setInput("Error");  
    }  
  };  
}
```

## 계산 입력/결과

2 + 2 = 4

1초 전



## History State 배열

"2 + 2 = 4"

history[0]

"5 \* 3 = 15"

history[1]

"10 / 2 = 5"

history[2]

"7 - 3 = 4"

history[3]

- 최근 계산 기록을 저장하고 표시하면 사용자 경험이 향상됩니다. 배열 형태로 저장하고 리스트 렌더링을 통해 화면에 출력합니다.

## = 버튼 클릭 시 기록 업데이트

■

```
// = 버튼 클릭 시 기록 추가  
if (value === "=") {  
try {  
const result =  
eval(input).toString();  
setInput(result);  
setHistory(prev => [  
input + " = " + result,  
...prev  
]);  
} catch {  
setInput("Error");  
}  
}
```

새 계산:  $2+3 = 5$



기존 기록 1:  $10-2 = 8$

기존 기록 2:  $4*3 = 12$

기존 기록 3:  $20/5 = 4$

**i** 배열의 맨 앞에 새 기록을 추가하고 기존 기록은 스프레드 연산자 (...prev)로 뒤로 이동

**!** 불변성(Immutability) 유지: 기존 배열 변경 대신 새 배열 생성



스프레드 연산자로 최신 기록이 맨 위로 오도록 배치

# History 출력

```
// App.js에 History 컴포넌트 추가  
import History from  
'./components/History';  
  
// 렌더링 부분에 History 추가  
return (  
  <div className="container">  
    <Display value={input} />  
    <Keypad onKey={handleClick} />  
    <History records={history} />  
  </div>  
);
```

## ⌚ 계산 기록

$2 + 3 =$

5

$10 * 5 =$

50

$100 / 4 =$

25

$32 - 12 =$

20

$5 * 5 =$

25



⌚ 최근 계산 기록이 위에 추가되고, 제한된 개수만 표시

- 1 History 컴포넌트 임포트
- 2 records 속성으로 history 배열 전달
- 3 기록이 없을 경우 조건부 렌더링 활용 가능

## 기록 개수 제한하기

```
// = 버튼 클릭 시 기록 업데이트 &  
제한  
const result =  
eval(input).toString();  
setInput(result);  
  
const newRecord = `${input}  
= ${result}`;  
setHistory(prev =>  
[newRecord,  
...prev].slice(0, 5)  
);
```

기록이 추가될 때마다 **최근 5개만 유지**

5+5 = 10

2\*3 = 6

10-4 = 6

8/2 = 4

7+1 = 8



새 계산 결과 추가 시 **slice(0, 5)** 적용

9-3 = 6

5+5 = 10

2\*3 = 6

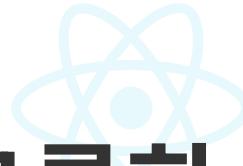
10-4 = 6

8/2 = 4

7+1 = 8

💡 최근 계산 기록 5개만 유지하여 UI를 깔끔하게  
관리

# Part F



## Keypad 그룹화 & 리팩토링

---

# Keypad 컴포넌트 구조

계산기의 주요 **컴포넌트 구조**는 다음과 같이 설계 되었습니다:

**App** - 최상위 컴포넌트, 전체 상태 관리

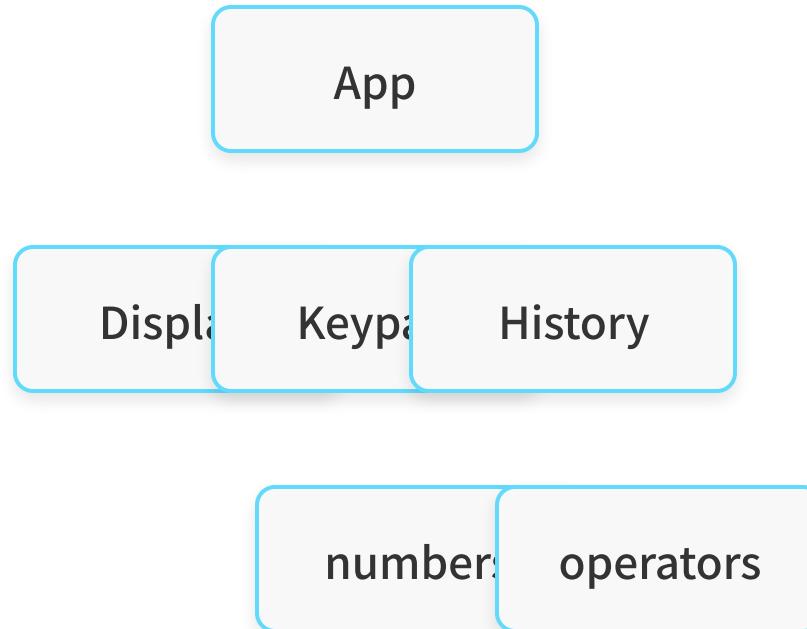
**Display** - 계산 결과 표시

**Keypad** - 숫자와 연산자 버튼 그룹화

- └ **numbers** - 숫자 버튼들
- └ **operators** - 연산자 버튼들

**History** - 계산 기록 표시

💡 컴포넌트 분리로 코드 재사용성과 유지보수성 향상



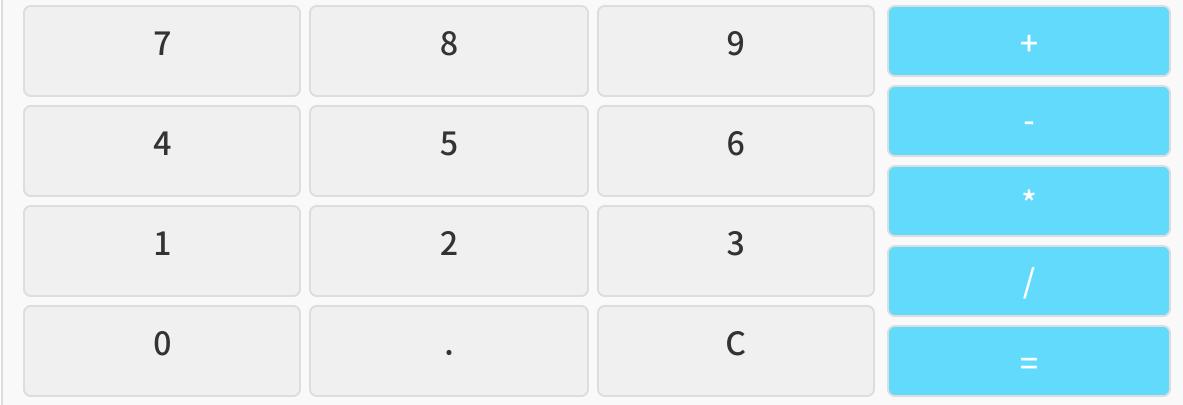
App ┌── Display ┌── Keypad | ┌── numbers (Button 컴포넌트들)  
| └── operators (Button 컴포넌트들) └── History

# Keypad 컴포넌트 구현 예시

```
// src/components/Keypad.jsx
export default function Keypad({ onKey }) {
  const numbers = ["7", "8", "9", "4", "5", "6",
    "1", "2", "3", "0", ".", "C"];
  const operators = ["+", "-", "*", "/", "="];
  return (
    <div className="keypad">
      <div className="numbers">
        {numbers.map(n => <Button
          key={n}
          label={n}
          onClick={onKey} />)
      </div>
      <div className="operators">
        {operators.map(o => <Button
          key={o}
          label={o}
          onClick={onKey} />)
      </div>
    </div>
  );
}
```

💡 숫자와 연산자 버튼을 그룹화하여 배치

## Keypad 컴포넌트 구조

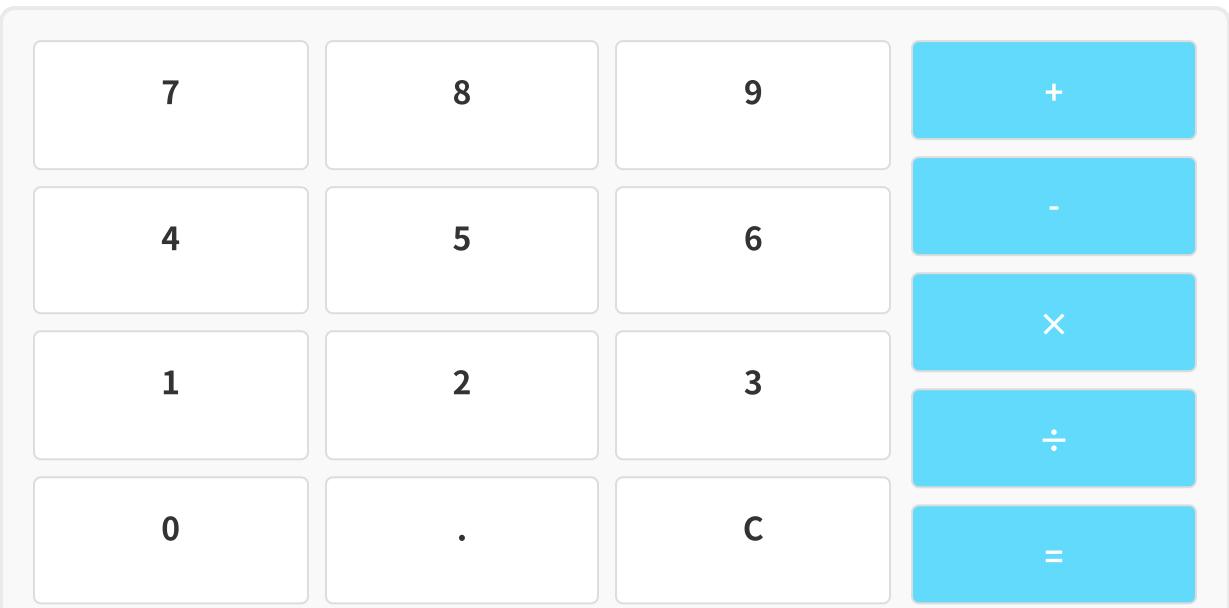


ℹ️ map()을 활용하여 반복 코드 최소화 및 유지보수성 향상

🔗 CSS Grid를 활용한 키패드 레이아웃 구현

# CSS 리팩토링

```
/* CSS Grid를 활용한 레이아웃 */  
keypad {  
display: grid;  
grid-template-columns: 1fr 80px;  
gap: 8px;  
}  
.numbers {  
display: grid;  
grid-template-columns: repeat(3, 1fr);  
gap: 8px;  
}  
.operators {  
display: grid;  
grid-auto-rows: 56px;  
gap: 8px;  
}
```



## CSS Grid 사용 이점

- ✓ 복잡한 계산기 레이아웃을 쉽게 구현
- ✓ 반응형 디자인 적용 용이
- ✓ 버튼 간격과 정렬 자동 조정
- ✓ 코드 가독성 향상과 유지보수 편리

💡 그리드 레이아웃으로 계산기 버튼을 직관적으로 배치

# App.js에서 Keypad, History 사용

```
// App.js
import { useState } from "react";
import Display from "./components/Display";
import Keypad from "./components/Keypad";
import History from "./components/History";
function App() {
  const [input, setInput] = useState("");
  const [history, setHistory] = useState([]);
  const handleClick = (value) => {
    // 버튼 처리 로직
  };
  return (
    <div className="container">
      <h1>React 계산기</h1>
      <Display value={input} />
      <Keypad onKey={handleClick} />
      <History records={history} />
    </div>
  );
}
```

props를 통해 자식 컴포넌트에 데이터와 함수를 전달합니다.

- Display: **value** prop으로 현재 입력값
- Keypad: **onKey** prop으로 이벤트 핸들러
- History: **records** prop으로 계산 기록 배열

## App 컴포넌트

(State: input, history)

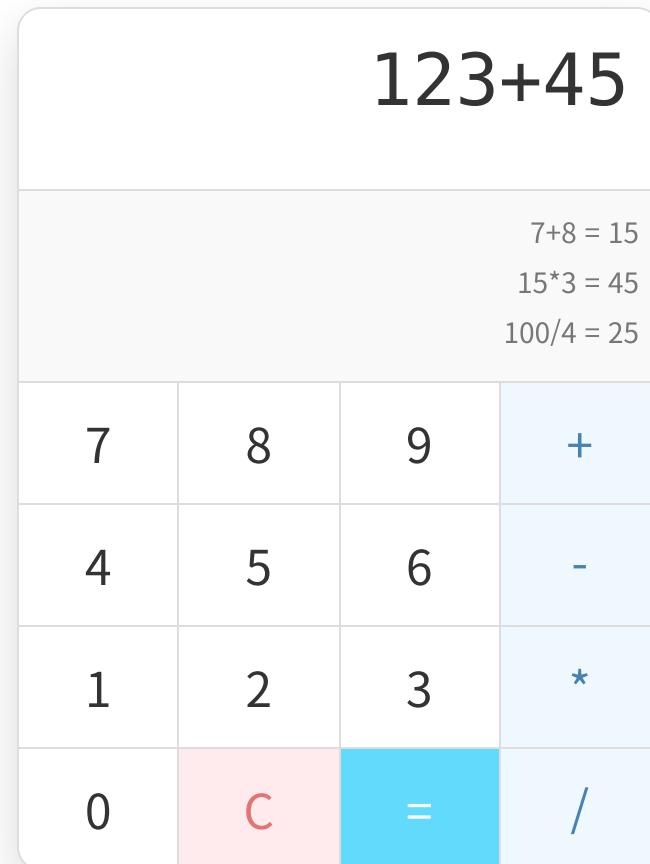


### 단방향 데이터 흐름 (Top-Down)

부모 컴포넌트에서 자식 컴포넌트로 props를 통해 데이터와 함수를 전달

# 완성된 계산기 앱 실행 화면

- 모든 컴포넌트가 통합된 최종 UI
- State를 활용한 입력값 및 결과 관리
- 계산 기록(History) 표시 기능
- 숫자와 연산자 Keypad 영역 분리
- CSS Grid로 깔끔한 버튼 배치



React 컴포넌트 기반 설계로 구현된 계산기 앱

# Part G



## 정리, 점검, 보강

---

# 오늘 배운 React 핵심



## State 관리

useState Hook으로 컴포넌트의 상태를 관리하고 업데이트



## 이벤트 처리

onClick 등의 이벤트 핸들러로 사용자 입력 처리



## 조건부 렌더링

조건에 따라 UI 요소를 선택적으로 표시



## 리스트 렌더링

배열 데이터를 map()으로 반복 처리하여 UI 생성



## 컴포넌트 분리

기능별로 독립된 컴포넌트를 만들어 재사용성 확보



## ?

# Quiz 1

문제: 버튼 배열을 map으로 출력하는 이유는?

```
const buttons =  
["7", "8", "9", "+", "4", "5", "6", "-  
", "1", "2", "3", "*", "0", "C", "=", "/"];  
  
{buttons.map((b, idx) => (  
  
    <Button key={idx} label={b} onClick=  
    {handleClick} />  
))}
```



정답

정답 보기

## ?

## Quiz 2

문제: 조건부 렌더링을 적용해 오류 메시지를 표시하는 코드를 작성하세요.

```
// input 상태가 "Error"일 때만  
// 빨간색으로 "잘못된 수식입니다."  
// 메시지를 표시하는 JSX 코드를 작성하세요.  
  
// input 상태:  
const [input, setInput] = useState("");  
// 여기에 조건부 렌더링 코드 작성
```



정답

정답 보기

## Quiz 3

문제: key={index} 대신 고유 id를 써야 하는 이유는?

```
// 인덱스를 key로 사용 (권장하지 않음)
{items.map((item, index) => (
  <li key={index}>{item.text}</li>
))}

// 고유 ID를 key로 사용 (권장)
{items.map(item => (
  <li key={item.id}>{item.text}</li>
))}
```



정답

정답 보기

## 학습 점검 체크리스트

React 계산기 앱 실습을 통해 학습한 핵심 개념들을 정확하게 이해했는지 체크해보세요. 각 항목에 자신 있게 '예'라고 답할 수 있다면 성공적으로 학습을 마친 것입니다!

### 학습 성공 팁

모든 항목에 '예'라고 답할 수 없다면, 해당 개념을 복습하거나 추가 실습을 통해 보완하세요. React의 기본기는 향후 더 복잡한 앱 개발의 토대가 됩니다.

### React 핵심 개념 체크리스트

- State로 입력값 관리**가 가능한가? - useState Hook을 사용해 계산기의 입력값과 결과값을 상태로 관리할 수 있습니까?
- 조건부 렌더링**을 적용할 수 있는가? - 조건에 따라 오류 메시지를 표시하는 등의 조건부 UI 렌더링을 구현할 수 있습니까?
- 리스트 렌더링**으로 기록을 출력할 수 있는가? - map 함수를 활용해 배열 데이터를 동적으로 렌더링하는 방법을 이해했습니까?
- 이벤트 처리**와 핸들러 함수를 작성할 수 있는가? - 버튼 클릭 등의 이벤트에 반응하는 핸들러 함수를 작성하고 연결할 수 있습니까?
- 컴포넌트 분리**와 props 전달 방식을 이해하는가? - 기능별로 컴포넌트를 분리하고 props를 통해 데이터와 함수를 전달할 수 있습니까?

# ☰ 과제 안내

오늘 학습한 계산기 앱을 확장해 보세요! 다음 과제들을 통해 React의 상태 관리와 컴포넌트 구조화 능력을 더 키울 수 있습니다.

📅 제출 기한: 다음 수업 전까지

## 필수 과제

- + % 연산 기능 추가하기 기본

예:  $200 \% 50 = 0$  (나머지 계산)

- + 기록 초기화 버튼 추가하기 기본

## ★ 선택 과제

- + 소수점 연산 정확도 개선하기 심화

- + 키보드 입력 지원 추가하기 심화

- + `math.js` 또는 `expr-eval` 라이브러리로 `eval()` 대체하기 도전

## 다음 주 예고

다음 주에는 [외부 API 연동 실습](#)을 통해 실제 서비스와 유사한 애플리케이션을 개발합니다.

React의 핵심 개념을 활용하여 외부 데이터를 효과적으로 가져오고 표시하는 방법을 배웁니다.

[useEffect Hook](#)과 [Fetch API](#)를 활용한 데이터 요청 패턴을 익힙니다.

### 날씨 API 연동

OpenWeatherMap API를 활용한 실시간 날씨 정보 앱

- ✓ 현재 위치 기반 날씨 정보 표시
- ✓ 5일 예보 차트로 시각화

### 영화 정보 API

TMDB(The Movie Database) API를 활용한 영화 검색 앱

- ✓ 인기 영화 목록 및 상세 정보 표시
- ✓ 키워드 기반 영화 검색 기능

### 사전 API

사전 API를 활용한 단어 검색 앱

- ✓ 단어 의미, 발음, 예문 표시
- ✓ 검색 기록 저장 및 관리

 준비물: OpenWeatherMap, TMDB 무료 API 키 발급 (수업 전 안내 예정)



## Q&A

어떤 질문이든 환영합니다!

함께 배우고 성장하는 시간이 되었길 바랍니다.

- ▣ **실습 관련** - React 계산기 구현, 코드 문제 해결
- ▣ **주제 이해** - State, 이벤트, 조건부 렌더링 등
- 🛡 **보안 이슈** - eval() 대안, 안전한 코드 작성법
- ✖ **확장 개발** - 기능 추가, 과제 관련 문의

다음 주 예고: 외부 API 연동 실습 (날씨, 영화, 사전)