React가 컴포넌트와 Hook을 호출하는 방식

React는 사용자 경험을 최적화하기 위해 필요할 때마다 컴포넌트와 Hook을 렌더링하는 역할을 합니다. React는 선언적입니다. 즉 컴포넌트의 로직에서 무엇을 렌더링할지를 React에 알려주면, React는 이를 사용자에게 가장 잘 표시할 방법을 찾아냅니다.

- 컴포넌트 함수를 직접 호출하지 마세요
- Hook을 일반 값처럼 전달하지 마세요
  - Hook을 동적으로 변경하지 마세요
  - Hook을 동적으로 사용하지 마세요

컴포넌트 함수를 직접 호출하지 마세요

컴포넌트는 JSX에서만 사용해야 합니다. 일반 함수처럼 호출하지 마세요. React가 호출해야 합니다.

React는 렌더링하는 동안 컴포넌트 함수가 언제 호출될지 결정해야 합니다. React는 이를 JSX로 수행합니다.

```
function BlogPost() {
  return <Layout><Article /></Layout>; // ✅ Good: 컴포넌트를 JSX에서만 사용합니다.
}
```

```
function BlogPost() {
  return <Layout>{Article()}</Layout>; // 🔴 Bad: 컴포넌트 함수를 직접 호출하지 마세요.
}
```

컴포넌트가 Hook을 포함하고 있다면, 컴포넌트를 반복문이나 조건문에서 직접 호출할 때 Hook의 규칙을 위반하기 쉽습니다.

React가 렌더링을 조정하도록 하면 여러 이점이 있습니다.

- **컴포넌트가 함수 이상의 역할을 하게 됩니다.** React는 Hook을 사용해 컴포넌트에 *지역 State*와 같은 기능을 추가하여, 컴포넌트가 트리 내에서 고유한 정체성을 갖도록 할 수 있습니다.
- **컴포넌트 타입이 재조정 과정에 참여합니다.** React가 컴포넌트를 호출하도록 하면 트리의 개념적 구조에 대해 더 많은 정보를 제공하게 됩니다. 예를 들어 `<Feed>` 에서 `<Profile>` 페이지로 전환될 때 React는 이를 재사용하려고 하지 않습니다.
- **React가 사용자 경험을 향상할 수 있습니다.** 예를 들어 React는 컴포넌트를 호출하는 사이에 브라우저가 일부 작업을 수행할 수 있도록 하여, 큰 컴포넌트 트리를 다시 렌더링하는 것이 메인 스레드를 차단하지 않도록 할 수 있습니다.
- **더 나은 디버깅 경험을 제공합니다.** 컴포넌트가 라이브러리에서 일급 객체로 취급되면, 개발 중에 분석할 수 있는 풍부한 개발 도구를 제공할 수 있습니다.
- **재조정 과정이 더 효율적입니다.** React는 트리에서 정확히 어떤 컴포넌트가 다시 렌더링이 필요한지 결정하고, 필요 없는 컴포넌트는 건너뛸 수 있습니다. 이는 앱을 더 빠르고 민첩하게 만듭니다.

Hook을 일반 값처럼 전달하지 마세요

Hook은 컴포넌트나 Hook 내부에서만 호출되어야 합니다. Hook을 일반 값처럼 전달하지 마세요.

Hook은 컴포넌트에 React 기능을 추가할 수 있게 합니다. Hook은 항상 함수로 호출되어야 하며 일반 값처럼 전달하면 안 됩니다. Hook을 함수로 호출해야, 개발자가 컴포넌트를 독립적으로 이해할 수 있는 *지역 추론* 이 가능합니다.

이 규칙을 어기면 React가 컴포넌트를 자동으로 최적화하지 못합니다.

Hook을 동적으로 변경하지 마세요

Hook은 가능한 한 “정적”으로 유지되어야 합니다. 이는 Hook을 동적으로 변경해서는 안 된다는 의미입니다. 예를 들어 고차 Hook을 작성하면 안됩니다.

```
function ChatInput() {
```

```
  const useDataWithLogging = withLogging(useData); // 🔴 Bad: 고차 Hook을 작성하지 마세요
  const data = useDataWithLogging();
}
```

Hook은 변경 불가능해야 하며 수정되지 않아야 합니다. Hook을 동적으로 변경하는 대신, 원하는 기능을 가진 정적인 Hook을 작성하세요.

```
function ChatInput() {
  const data = useDataWithLogging(); // ✅ Good: Hook을 새로 작성하세요.
}

function useDataWithLogging() {
  // ... 여기에 새로운 Hook의 로직을 작성하세요
}
```

Hook을 동적으로 사용하지 마세요

Hook은 동적으로 사용해서도 안 됩니다. 예를 들어 Hook을 값으로 전달하여 컴포넌트에 의존성을 주입하는 대신,

```
function ChatInput() {
  return <Button useData={useDataWithLogging} /> // 🔴 Bad: Hook을 Props로 전달하지 마세
}
```

항상 Hook 호출을 해당 컴포넌트에 직접 작성하고, 모든 로직을 그 안에서 처리해야 합니다.

```
function ChatInput() {
  return <Button />
}

function Button() {
  const data = useDataWithLogging(); // ✅ Good: Hook을 직접 사용하세요.
}

function useDataWithLogging() {
  // Hook의 동작을 변경하는 조건부 로직이 있다면, 이는 Hook 내부에 포함해야 합니다.
```

```
}
```

이렇게 하면 `<Button />` 컴포넌트가 훨씬 이해하기 쉽고 디버깅하기도 쉬워집니다. Hook을 동적으로 사용하면 앱의 복잡성이 크게 증가하고, 지역 추론을 방해하여 장기적으로 팀의 생산성을 저하시킵니다. 또한 Hook을 조건부로 호출하면 안된다는 Hook의 규칙을 실수로 어기기 쉽습니다. 테스트를 위해 컴포넌트를 모킹Mocking해야 한다면, 대신 서버를 모킹Mocking하여 미리 준비된 데이터에 응답하도록 하는 것이 낫습니다. 가능하다면 앱을 end-to-end 테스트하는 것이 더 효과적입니다.

이전
컴포넌트와 Hook은 순수해야 합니다

다음
Hook의 규칙