4 Hooks

⭐ **사전과제**

필수
1. useState 문법(2가지 방법)
2. useEffect와 컴포넌트 생명주기

선택
1. Hook의 동작원리

노션, 블로그, 깃허브 등등 어떠한 방식으로도 정리 가능하며 해당 링크를 웹 파트 노션의 4주차 과제란에 올려주세요.
4주차 웹 세션 시작전 1~2분을 무작위로 선정하여 약 3분간의 공부한 내용에 대한 발표를 부탁드릴예정입니다!

- useState : https://ko.legacy.reactjs.org/docs/hooks-state.html

사전과제 참고 자료

- useEffect : https://ko.legacy.reactjs.org/docs/hooks-effect.html
- Hook 동작원리 : https://hewonjeong.github.io/deep-dive-how-do-react-hooks-really-work-ko/

useEffect

💡 react에서 **side effect**를 수행할 수 있도록 해주는 Hook

Side Effect란?

리액트 컴포넌트의 외부에서 일어나는 작업들을 말한다.

ex) localStorage와 같은 브라우저API, 서버 API를 통한 요청 등

왜 Side Effect 실행을 별도로 구분할까?

Side Effect 중 서버에서 많은 양의 데이터를 받아오는 작업을 수행한다고 가정하자.

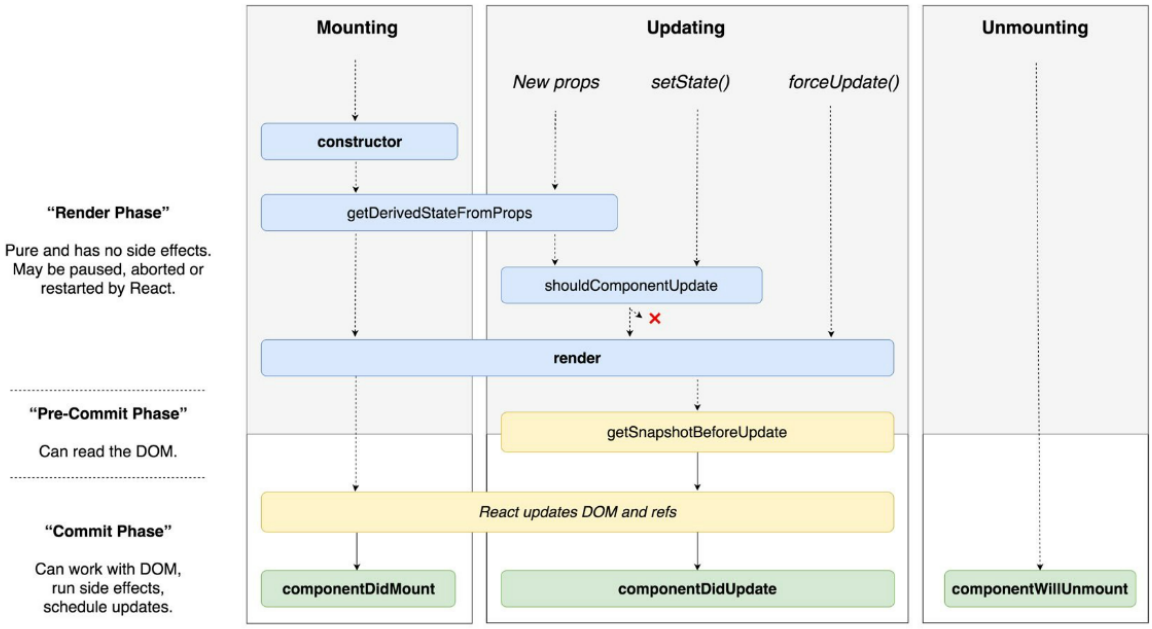
useEffect를 이용하지 않고 위의 코드를 실행한다고 했을 때,

리렌더링이 될 때마다 해당 작업을 수행해서 프로그램 성능을 저하시킬 것이다.

때문에 **Side Effect가 필요할 때만 해당 작업을 수행**할 수 있도록 구분하는 것이다.

> Side Effect가 필요할 때 특정 작업을 수행하려면 **컴포넌트의 생명 주기**에 대해 알아야 한다.

컴포넌트 생명 주기



Mount

처음 컴포넌트가 화면에 나타날 때

Render

처음 컴포넌트가 나타날 때와 변경사항이 있을 때

컴포넌트가 리렌더링 될 때

- **props**가 변경될 때
- **부모 컴포넌트**가 리렌더링될 때
- **state**값이 변할 때

Unmount

컴포넌트가 화면에서 사라질 때

해당 컴포넌트에서 구독했던 API 등을 해제하는 **뒷정리 함수를 실행**한다.

useEffect 사용

기본 문법

```
useEffect(function, [deps]);
```

- **function** : 특정한 순간에 실행하고자 하는 side effect 함수
- **deps** : 의존성 배열
  - deps가 없을 경우는 **렌더링 될 때마다 실행**
  - **특정 state가 변경될 때마다** 위 함수를 실행
  - 빈 배열일 경우 **mount 됐을 때만** 실행

컴포넌트 생명 주기 실습

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

const App = () => {
  const [text, setText] = useState("");
  const [counter, setCounter] = useState(0);

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

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

  useEffect(() => {
    console.log("counter change!");
  }, [counter]);

  const handleCounter = () => {
    setCounter(counter + 1);
  };

  const handleText = (e) => {
    setText(e.target.value);
  };
  return (
    <>
      <div>
        <input onChange={handleText} value={text} />
      </div>
      <div>
        <h1>{counter}</h1>
        <button onClick={handleCounter}>+1</button>
      </div>
    </>
  );
};

export default App;
```

---

useState

💡 react에서 상태 관리를 도와주는 Hook

왜 로컬 변수로 상태 관리를 하면 안 될까?

```
function App() {
  let count = 0;
  const handleClick = () => {
    count++;
  };
  return (
    <div>
      <h1>{count}</h1>
      <button onClick={handleClick}>+1</button>
    </div>
```

```
  );
}

export default App;
```

- 로컬 변수는 리렌더링 시 **값이 유지되지 않는다.**
- 로컬 변수의 변화 값은 **리렌더링을 야기하지 않는다.**

useState의 역할

- state값에 대해 리렌더링이 일어나더라도 **값을 유지**해야 한다.
- **리렌더링을 야기**해야 한다.

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

const App = () => {
  const [count, setCount] = useState(0);
  const handleClick = () => {
    setCount(count + 1);
  };
  return (
    <>
      <h1>{count}</h1>
      <button onClick={handleClick}>+1</button>
    </>
  );
};

export default App;
```

useState 사용

기본 문법

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

- **state** : **현재 상태 값**을 가지는 변수, 초기값은 initialState의 값
- **setState** : **state를 수정**할 수 있는 함수

❓ **배열 비구조화 할당**

위의 예시에서 useState는 아래와 같은 배열을 반환한다.
[S, Dispatch<SetStateAction<S>>]

위 배열의 값들을 변수에 할당하고 싶다고 했을 때
배열을 해체해서 그 값을 각각 변수에 담을 수 있게 하는 문법

setState

1. **새로운 값**으로 업데이트

```
setState(newState);
```

2. **함수형** 업데이트

```
setState((prev)=>prev+1)
```

- 현재 **state값을 전달 받고, 반환**해줌으로써 값을 수정한다.

특징

1. **Batch Update**

성능을 위해 **업데이트 작업을 일괄적으로 처리**해서 리렌더링 횟수를 최소화하는 기능

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

const App = () => {
  const [count1, setCount1] = useState(0);
  const [count2, setCount2] = useState(0);
  const [count3, setCount3] = useState(0);

  useEffect(() => {
    console.log("render!");
  }, [count1, count2, count3]);

  const handleClick = () => {
    setCount1(count1 + 1);
```

```
    setCount2(count2 + 1);
    setCount3(count3 + 1);
  };

  return (
    <>
      <h1>{count1}</h1>
      <h1>{count2}</h1>
      <h1>{count3}</h1>
      <button onClick={handleClick}>+1</button>
    </>
  );
};

export default App;
```

성능을 개선해주지만 **새로운 값으로 수정하는 방식을 이용했을 때**,

같은 state에 대해 동시에 여러 번 수정 작업을 했을 때 **마지막 수정 작업만 처리**된다.

함수형 업데이트로 위 문제를 해결할 수 있다.

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

const App = () => {
  const [count, setCount] = useState(0);

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

  const handleClick = () => {
    setCount((prev) => prev + 1);
    setCount((prev) => prev + 1);
    setCount((prev) => prev + 1);
  };

  return (
    <>
      <h1>{count}</h1>
      <button onClick={handleClick}>+1</button>
    </>
  );
};

export default App;
```

2. **setState의 비동기 처리**

batch update를 하기 때문에 수정 작업이 동기적으로 처리될 수 없다.

→수정 작업 이후 수정된 state를 이용한 동기적인 작업에서 **변경된 state값을 사용할 수 없다.**

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

const UseState = () => {
  const [count, setCount] = useState(0);

  const handleClick = () => {
    setCount((prev) => prev + 1);
    alert(count);
  };

  return (
    <>
      <h1>{count}</h1>
      <button onClick={handleClick}>+1</button>
    </>
  );
};

export default UseState;
```

useEffect의 의존성 배열로 위 문제를 해결할 수 있다.

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

const UseState = () => {
  const [count, setCount] = useState(0);

  useEffect(() => {
    alert(count);
  }, [count]);

  const handleClick = () => {
    setCount((prev) => prev + 1);
  };

  return (
    <>
      <h1>{count}</h1>
      <button onClick={handleClick}>+1</button>
    </>
  );
};

export default UseState;
```

---

useContext

> 💡 **Context를 사용**할 수 있게 도와주는 Hook

전역 상태 관리?

여러 컴포넌트에서 사용하는 값은 보통 props로 전달한다.

하지만 컴포넌트의 deps가 깊어지게 되면

사용하지도 않는 props를 오직 전달하기 위해서 받아오고 전달하는 경우가 있다. (통로가 된 것처럼)

이를 Props Drilling이라고 한다.

이런 문제를 해결하기 위해 **전역적으로 값을 공유**할 수 있도록 해주는 것이 **Context**이다.

useContext는 생성된 **Context를 사용할 수 있게 도와주는 Hook**이다.

Context 생성

```
const SomeContext = createcContext(defaultValue);
```

생성된 SomeContext는 지정된 범위 내에서 props로 따로 전달되지 않아도 바로 사용될 수 있다.

Context.Provider

생성된 Context를 어떤 범위 내에서 사용할지 Provider를 통해 지정할 수 있다.

```
<SomeContext.Provider value={someValue}>
    <ChildComponent/>
</SomeContext.Provider>
```

위처럼 **Provider로 감싼 자식 컴포넌트**에서는 SomeContext값을 이용할 수 있다.

즉 Context.Provider 컴포넌트는 **Context 사용 가능한 범위를 본인의 자식 컴포넌트로 지정**하는 것이다.

useContext

useContext는 **지정된 범위** 내에서 **Context 객체를 가져옴**으로써

그 **값을 사용**할 수 있도록 해주는 Hook인 것이다.

```
const contextValue=useContext(SomeContext);
```

위처럼 SomeContext를 전달해서 그 값을 사용할 수 있다.

실습

📢 자기소개 페이지에 로그인 구현

조건

- 맨 처음 **localStorage에 userInfo 데이터를 삽입**
- App.js 컴포넌트가 렌더링 될 때 **isLogin state 값**으로 login이 되었는지 확인
  - isLogin의 초기 값은 false
- App.js에서 isLogin이 **false면 로그인 화면**, **true면 자기소개 페이지** 렌더링
- 로그인 시 **입력값이 localStorage에 있는 값과 동일할 경우 isLogin값을 true**로 바꾼다.
  - 틀렸을 경우 **alert** 띄워주기

```
//UserInfo.js
export const UserInfo = {
  id: "choi",
```

```
  password: "test",
};
```

```
//Login.js
import React, { useState } from "react";
import "./Login.css";

const Login = () => {
  const [id, setId] = useState("");
  const [password, setPassword] = useState("");

  return (
    <div className="wrapper_box">
      <h2>LOG IN</h2>
      <form className="login_form">
        <input type="text" className="login_input" placeholder="ID" />
        <input placeholder="Password" type="Password" className="login_input" />
        <button type="submit" className="login_btn">
          Login
        </button>
      </form>
    </div>
  );
};

export default Login;
```

```
//Login.css
.wrapper_box {
  margin: 150px auto;
  display: flex;
  flex-direction: column;
  align-items: center;
  color: #7ac142;
}

.login_form {
  display: flex;
  flex-direction: column;
  width: 30%;
  height: 300px;
  align-items: center;
}

.login_input {
  width: 300px;
  height: 50px;
  border: none;
  border-bottom: 1px solid gray;
  outline: none;
}

.login_input:focus {
```

```
  border-bottom: 1px solid #7ac142;
}

.login_btn {
  width: 120px;
  height: 30px;
  text-align: center;
  margin-top: 20px;
  cursor: pointer;
  background-color: #7ac142;
  color: white;
  border-radius: 20px;
  border: none;
}
```

과제

- 로그인 페이지 구현 못한 것 마저 하기
- 오늘 강의 내용 중 **추가로 궁금한 부분 2개 이상** 정리하기
  - 없다면 useState가 어떻게 렌더링을 야기하는지(간단하게)정리, 오늘 다루지 않은 Hook 정리
- 본인 자기 소개 페이지에 다크 모드 적용
  - useContext를 이용해서 다른 자식 컴포넌트들도 다크 모드가 적용될 수 있도록

회원들에게 제공할 템플릿 → 혜민 의견

css 파일 + 아래 파일

```
//Login.js
import React, { useState } from "react";
import "./Login.css";

const Login = ({ setIsLogin }) => {
  const [id, setId] = useState("");
  const [password, setPassword] = useState("");

  return (
    <div className="wrapper_box">
      <h2>LOG IN</h2>
```

```
      <form className="login_form" >
        <input
          type="text"
          className="login_input"
          placeholder="ID"
        />
        <input
          placeholder="Password"
          type="password"
          className="login_input"
        />
        <button type="submit" className="login_btn">
          Login
        </button>
      </form>
    </div>
  );
};

export default Login;
```