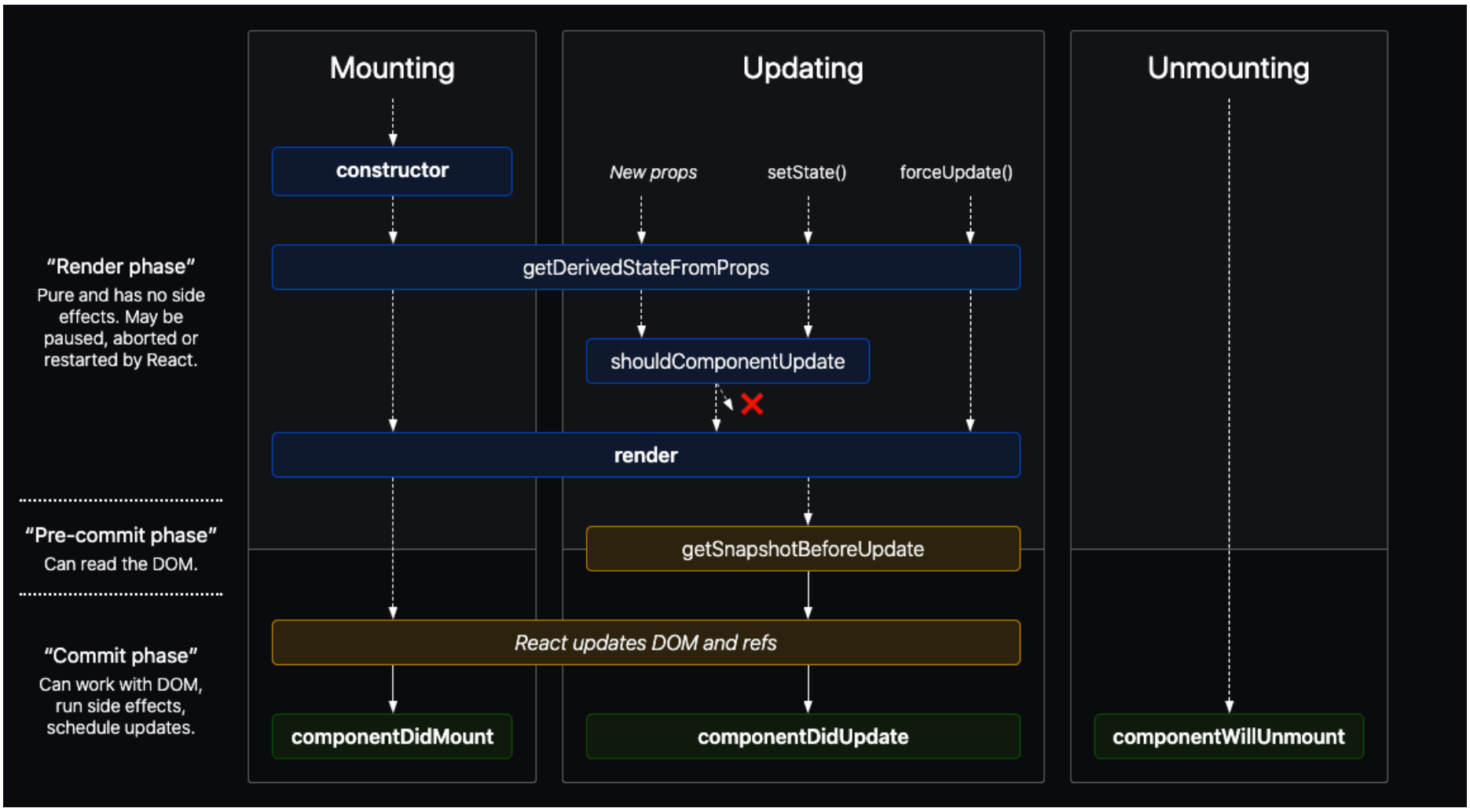
# 라이프 사이클

| | |
|---|---|
| Status | Not started |
| Assign | React.js |

## 1. class에서의 생명 주기




ref : projects.wojtekmaj.pl/react-lifecycle-methods-diagram/

### #라이프 사이클에 대한 설명 1

### component가 mount 시작 되면

1. constructor가 가장 먼저 실행
2. getDerivedStateFromProps에서 props와 state를 비교
3. render component
4. update DOM
5. componentDidMount가 실행

### component가 update 될 때 (new props or setState)

1. getDerivedStateFromProps에서 props와 state 비교
2. shouldComponentUpdate 실행 - new props, state를 확인하고 rerender 할 것인지 결정
3. rerender
4. update DOM
5. componentDidUpdate 실행

## component가 unmount 될 때

1. componentWillUnmount 실행

## #라이프 사이클에 대한 설명 2

## 1. 마운트 ( 생성 )

컴포넌트의 인스턴스가 생성되어, DOM에 삽입될 때 순서대로 호출된다.

1. **constructor()** :컴포넌트를 새로 만들 때마다 호출되는 클래스 생성자 메서드`this.props`, `this.state`에 접근할 수 있으며 리액트 요소를 반환한다.setState를 사용할 수 없으며 DOM에 접근해선 안된다.
2. **getDerivedStateFromProps()** :props에 있는 값을 state에 동기화 시킬 때 사용하는 메서드
3. **render()** :UI를 렌더링하는 메서드
4. **componentDidMount()** :컴포넌트가 웹 브라우저 상에 나타난 후 즉 첫 렌더링을 마친 후에 호출하는 메서드라이브러리나 프레임워크의 함수를 호출하거나 이벤트 등록, `setTimeout`, `setInterval`과 같은 **비동기 작업**을 처리하면 되고, `setState` 호출도 이 메서드에서 호출하는 경우가 많다.

## 2. 업데이트

props나 state가 변경되면 렌더가 진행되며 순서대로 호출된다.

1. **getDerivedStateFromProps()** :이 메서드는 마운트 과정에서 호출되며, 업데이트가 시작하기 전에도 호출된다.props의 변화에 따라 state 값에도 변화를 주고 싶은 경우에 사용한다.
2. **shouldComponentUpdate()** :props또는 state를 변경했을 때, 리렌더링을 시작할지 여부를 지정하는 메서드true를 반환하면 다음 라이프사이클 메서드를 계속 실행하고,false를 반환하면 작업을 중지한다.
3. **render()** :컴포넌트 리렌더링
4. **getSnapshotBeforeUpdate()** :컴포넌트 변화를 DOM에 반영하기 바로 직전에 호출하는 메서드
5. **componentDidUpdate()** :컴포넌트 업데이트 작업이 끝난 후 호출하는 메서드.

## 3. 언마운트 ( 마운트 해제 )

컴포넌트를 DOM에서 제거하는 과정

1. **componentWillUnmount()** :컴포넌트를 DOM에서 제거할 때 실행한다.이후에 컴포넌트는 다시 렌더링 되지 않으므로, 여기에서 setState()를 호출하면 안된다.

### 요약

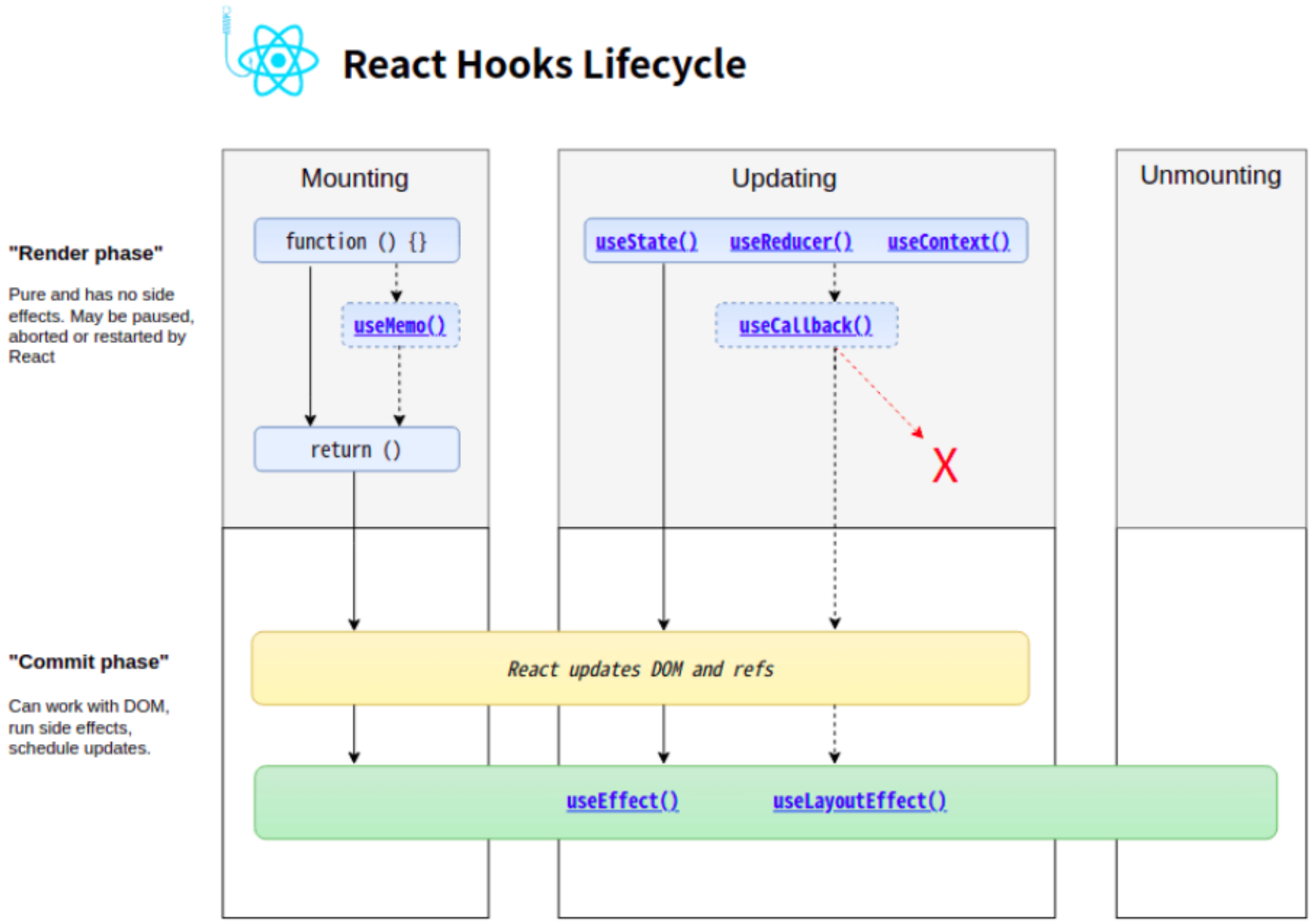
최초 : constructor -> getDerivedStateFromProps -> componentDidMount

업데이트 : getDerivedStateFromProps -> componentDidUpdate

언마운트 : componentWillUnmount

## 2. Function에서 생명주기, Hook에서는 useEffet()가 위 생명주기 함수들을 대체한다.

# React Hooks에서의 생명주기




## useEffect의 기본 형태

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

- useEffect는 기본적으로 모두 합니다.

  **componentDidMount, componentDidUpdate, componentWillUnmount, getDerivedStateFromProps의 역할**때문에 위 코드는

  1) 컴포넌트가 마운트 된 후, 2) 컴포넌트가 업데이트되고 난 후 3) 컴포넌트가 언마운트 되기 전 모두 실행됩니다.

- useEffect는 아래와 같이 2가지의 인수를 받습니다. 하나는 useEffect가 실행할 함수이며, 두 번째는 이 함수를 실행시킬 조건입니다.

```
useEffect(<function>, <array>);
```

- useEffect는 함수를 반환할 수 있습니다. 이를  이라고 표현합니다.

  **clean-up**

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

- 이 clean-up함수는 useEffect가 실행될 때마다 실행됩니다.

**ComponentDidUpdate or getDerivedStateFromProps**

- 위에서 useEffect는 1) 컴포넌트가 마운트 된 후, 2) 컴포넌트가 업데이트되고 난 후 3) 컴포넌트가 언마운트 되기 전 모두 실행된다고 했습니다.

- 이 때,  조건에 특정 state 혹은 props를 전달하면 이 state 혹은 props가 변할 때만 useEffect가 실행됩니다.
- 이는 class에서 **componentDidUpdate,  getDerivedStateFromProps와 같은 역할 입니다.**

```
cosnt { exampleProp } = props;
const [count, setCount] = useState(0);

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

**ComponentDidMount**

- 위에서 조건으로 state를 전달하면 그 state가 변화할 때만 useEffect가 실행된다고 했습니다.
- 따라서, 이 배열을 빈 배열로 전달하면 useEffect는 컴포넌트 마운트시에만 실행됩니다.
- 이는 class에서 **componentDidMount와 같은 역할 입니다.**

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

**ComponentWillUnmount**

- ComponentWillUnmonut의 역할은 **clean-up 함수를 통해 구현**할 수 있습니다.
- 컴포넌트가 Unmount될 때 정리하거나 unscribe 해야할 것이 있다면 useEffect의 return 값으로 전달합니다.

```
useEffect(()=>{
  console.log("");
    return(() => exampleAPI.unsubscribe());
})
```

## 리액트 훅을 도입한 목적

- Hook을 활용하면 **상태 관련 로직을 추상화해 독립적인 테스트와 재사용이 가능해 레이어 변화 없이 재사용**할 수 있다.
- 기존의 라이프 사이클 메서드 기반이 아닌 **로직 기반**으로 나눌 수 있어서 컴포넌트를 함수 단위로 잘게 쪼갤 수 있다는 이점 때문이다