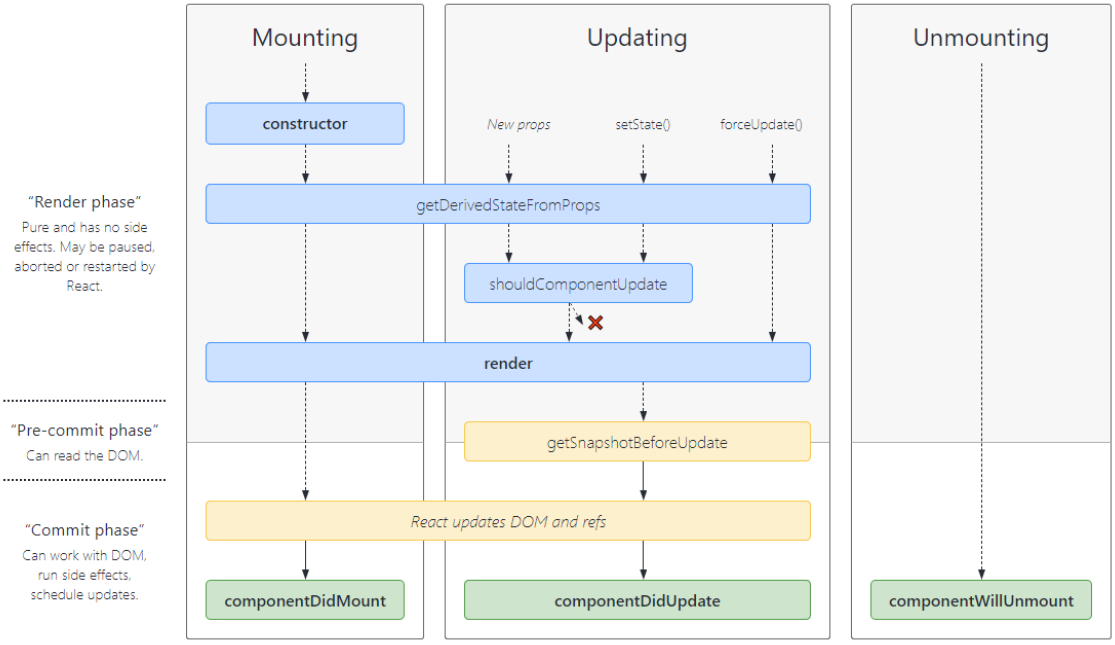
7. Component Life Cycle

- 모든 리액트 컴퍼넌트에는 Life Cycle이 존재한다.



7.1 Life Cycle Method

- 라이프 싸이클 메서드는 총 9개이다.
- `will` 접두사가 붙은 메서드는 특정 작업의 시작전에 실행 되는 메서드이고 `did` 접두사가 붙은 메서드는 작업 종료후 실행 되는 메서드이다.
- 라이프 싸이클은 총 세가지 즉, `마운트 업데이트, 언마운트` 카테고리로 나눈다.

7.1.1 mount

- dom이 생성되고 웹브라우저상에 나타나는 것을 마운트(mount)라고 한다.
- 마운트할 때 호출하는 메서드

> `constructor > getDeriverdStateFromProps > render > componentDidMount`
>
> - constructor : 컴퍼넌트를 새로 생성할 때마다 호출되는 클래스 생성자 메서드
> - getDeriverdStateFromProps : props에 있는 값을 state에 넣을 때 사용하는 메서드
> - render : UI를 랜더링하는 메서드
> - componentDidMount : 컴퍼넌트가 웹브라우저에 나타난 후에 호울하는 메서드

7.1.2 update

- 컴퍼넌트는 다음과 같은 총 4가지 경우에 업데이트한다.
  1. props가 바뀔 때
  2. state가 바뀔 때
  3. 부모 컴퍼넌트가 리랜더링될 때
  4. this.forceUpdate로 강제로 랜더링을 트리거 할 때
- 업데이트할 때 호울하는 메서드

> `getDerivedStateFromProps > shouldComponentUpdate > getSnapshotBeforeUpdate > componentDidUpdate`
>
> > - getDerivedStateFromProps : 마운트과정에서도 호출, 업데이트시작 전에 호출되고 props변경에 따라 state값에도 변경할 때 사용
> > - shouldComponentUpdate : 컴퍼넌트가 리렌더링 여부를 결정하는 메서드, 이 메서드는 true, false를 리턴한다.
> >   - true를 반환하면 다음 라이프사이클레서 메서드를 계속 실행하고
> >   - false를 반환하면 작업을 중지 즉, 컴퍼넌트가 리랜더링되지 않는다.
> >   - 만약 특정함수에서 this.forceUpdate()함수를 호출한다면 이 과정은 생략하고 바로 render함수를 호출한다.
> > - getSnapshotBeforeUpdate : 컴퍼넌트 변화를 DOM에 반영하기 바로 직전에 호출되는 메서드
> > - componentDidUpdate : 컴퍼넌트의 업데이트 작업이 끝난 후 호출되는 메서드

7.1.3 unmount

- 마운트의 반대과정 즉, 컴퍼넌트를 DOM에서 제거하는 것을 unmount라고 한다.
- 언마운트할 때 호출되는 메서드

> `componentWillUnmount`
>
> > - componentWillUnmount : 컴퍼넌트가 웨브라우저상에서 사라지기 전에 호출되는 메서드

7.2 Life Cycle Method 정리

7.2.1 render()함수

> `render() { ... }`
>
> - 이 메서드는 컴퍼넌트의 모양을 정의
> - 이 메서드안에서 `this.props와 this.state에 접근` 할 수 있으며, `리액트 요소를 반환` 한다.
> - 아무것도 보여주고 싶지 않을 경우 null 또는 false를 반환

> - 이 메서드 안에서는
>   - 이벤트 설정이 아닌 곳에서 setState를 사용불가
>   - 브라우저의 DOM에 접근 불가
>   - DOM의 정보를 가져오거나 state를 변경할 때는 componentDidUpdate에서 처리 해야 한다.

7.2.2 constructor 메서드

```
constructor(props) { ... }
```

- 컴퍼넌트의 생성자 메서드로 컴퍼넌트를 생성할 때 처음으로 실행
- 이 메서드에서는 state초기값을 설정 할 수 있다.

7.2.3 getDeriverdStateFromProps 메서드

- 리액트 v16.3이후에 생긴 메서드
- props에서 받아 온 값을 state에 동기화 시키는 용도로 사용
- 컴퍼넌트가 마운트될 떄와 업데이트될 때 호출 된다.

```
static getDeriverdStateFromProps(nextProps, prevState) {
    if(nextProps.value !== prevState.value) { // 조건별로 특정 값을 동기화
        return { value: nextProps.value };
    }
    return null; // state를 변경할 필요가 없을 경우 null을 반환
}
```

7.2.4 componentDidMount메서드

```
componentDidMount() { ... }
```

- 이 메서드는 컴퍼넌트를 만들고 첫 랜더링을 종료 후 실행 한다.
- 이 메서드 안에서 다른 JavaScript 라이브러리, 프레임워크의 함수호출, 이벤트등록,
- setTimeout, setInterval, 네트워크요청과 같은 비동기 작업을 처리

7.2.5 shouldComponentUpdate 메서드

```
shouldComponentUpdate(nextProps, nextState) { ... }
```

- props or state를 변경했을 떄 리랜더링의 시작여부를 지정하는 메서드 이다.
- 이 메서드는 반드시 true or false값을 리턴 해야 한다.
- 컴퍼넌트 생성시 이 메서드를 별도로 생성하지 않으면 기본값은 언제나 true 이다
- false를 반환하면 업데이트과정은 여기서 중단된다.
- 이 메서드안에서 현재의 props와 state는 this.props와 this.state로 접근 하고
- 새로 생성된 props와 state는 nextProps와 nextState로 접근 할 수 있다.

7.2.6 getSnapshotBeforeUpdate 메서드

- 리액트 v16.3이후부터 적용

- 이 메서드는 `render에서 만들어진 결과물이 브라우저에 실제로 반영된기 직전에 호출`
- 반환하는 값은 `componentDidMount에서 세 번째 파라미터인 snapshot값으로 부터 전달` 받을 수 있다.
- 주로 `업데이트하기 직전의 값을 참고할 일이 있을 때 활용`
- 아래 예처럼 스크롤방의 위치등...

```
static getSnapshotBeforeUpdate(prevProps, prevState) {
    if(prevProps.array !== this.state.array) {
        const { scrollTop, scrollHeight } = this.list
        return { scrollTop, scrollHeight };
    }
}
```

7.2.7 componentDidUpdate 메서드

- `랜더링 완료 후에 실행`
- 업데이트가 종료 직후이기 때문에 `DOM관련 처리가 가능` 하다.
- prevProps, prevState를 사용하여 컴퍼넌트가 이전에 가졌던 데이터에 접근 가능
- getSnapshotBeforeUpdate에서 반환값이 있다면 snapshot 값을 전달 받을 수 있다.

```
componentDidUpdate(prevProps, prevState, snapshot) { ... }
```

7.2.8 componentWillUnmount 메서드

- 이 메서드는 `컴퍼넌트를 DOM에서 제거할 때 실행` 한다.
- componentDidMount메서드에서 등록한 event, timer, 직접 생성한 DOM이 있다면 여기서 제거작업을 해야 한다.

```
componentWillUnmount() { ... }
```

7.2.9 componentDidCatch메서드

- 컴퍼넌트 `랜더링 도중에 에러가 발생했을 때 애플리케이션이 먹통이 되지 않고 오류 UI를 보여 줄 수 있게 한다`
- error는 어떤 에러가 발생했는지를, info는 어디에서 에러가 발생했는지에 대한 정보를 제공
- 이 메서드를 사용할 때는 `컴퍼넌트 자신에게 발생하는 에러는 잡아낼 수 없고`
- 자신의 `this.props.children`으로 전달되는 컴퍼넌트에서 발생하는 에러만 잡아낼 수 있다.

```
componentDidCatch(error, info) {
    this.setState({
        error: true
    });
    console.log({ error, info });
}
```

7.3 Life Cycle 메서드 사용하기

7.3.1 예제컴퍼넌트 생성

```
src/App.js
import React, { Component } from 'react';
import React07LifeCycle01 from './mysrc/React07LifeCycle01';

// 랜덤 색상을 생성합니다.
function getRandomColor() {
  return '#' + Math.floor(Math.random() * 16777215).toString(16);
}

class App extends Component {

  state = {
    color: '#000000'
  };

  handleClick = () => {
    this.setState({
      color: getRandomColor()
    });
  };

  render() {
    return (
      <div>
        <button onClick={this.handleClick}>랜덤 색상</button>
        {/* React07LifeCycle01에 color값을 props로 전달 */}
        <React07LifeCycle01 color={this.state.color} />
      </div>
    );
  }
}
export default App;

src/mysrc/React07LifeCycle01.js
import { Component } from 'react';

// 이 컴퍼넌트는 각 사이클 메서드를 실행할 때마다 콘솔에 출력
// 부모 컴퍼넌트에서 props로 색상을 전달 받아 버튼클릭시 1씩 추가
// getDerivedStateFromProps는 부모에게 전달 받은 color값을 state에 동
기화
// getSnapshotBeforeUpdate는 DOM변화가 일어나기 직전의 color속성을
snapshot값
// 으로 반환하여 이것을 componentDidUpdate에서 조회
// shouldComponentUpdate에서 state.number 값의 마지막 자리수가 4이면
리랜더링을 취소

class React07LifeCycle01 extends Component {

  state = {
    number: 0,
    color: null,
  }

  myRef = null;
```

```
  constructor(props) {
    super(props);
    console.log('1. 생성자함수 호출!!')
  }

  static getDerivedStateFromProps(nextProps, prevState) {
    console.log("2.getDerivedStateFromProps메서드 호출!!");
    if(nextProps.color !== prevState.color) {
      return { color: nextProps.color };
    }
    return null;
  }

  componentDidMount() {
    console.log("3. componentDidMount 호출!! ");
  }

  shouldComponentUpdate(nextProps, nextState) {
    console.log("4. shouldComponentUpdate 호출!! ", nextProps,
nextState);
    return nextState.number % 10 !== 4;
  }

  componentWillUnmount() {
    console.log('5. componentWillUnmount 호출!!');
  }

  handleClick = () => {
    this.setState({
      number: this.state.number + 1,
    });
  };

  getSnapshotBeforeUpdate(prevProps, prevState) {
    console.log('6. getSnapshotBeforeUpdate 호출!!');
    if (prevProps.color !== this.props.color) {
      return this.myRef.style.color;
    }
    return null;
  }

  componentDidUpdate(prevProps, prevState, snapshot) {
    console.log('7. componentDidUpdate 호출!!', prevProps,
prevState);
    if (snapshot) {
      console.log('업데이트되기 직전 색상: ', snapshot);
    }
  }
  render() {
    console.log('8. render 호출!!');
    const style = {
      color: this.props.color,
    };
    return (
      <div>
        {/* {this.props.missing.value} */}
```

```
        <h1 style={style} ref={(ref) => (this.myRef = ref)}>
          {this.state.number}
        </h1>
        <p>color: {this.state.color}</p>
        <button onClick={this.handleClick}>더하기</button>
      </div>
    );
  }
}

export default React07LifeCycle01;
```

> - React.StrictMode가 적용되어 있으면 일부 라이프사이클 메서드가 두 번씩 호출된다.
> - 개발환경에서만 두 번씩 호출되며 프로덕션환경에서는 한 번만 호출된다.
> - 한 번만 호출하게 할 결우는 React.StrictMode를 제거하고 App 컴퍼넌트만 랜더링하면 된다.

```
index.js
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);

reportWebVitals();
```

7.3.2 에러처리

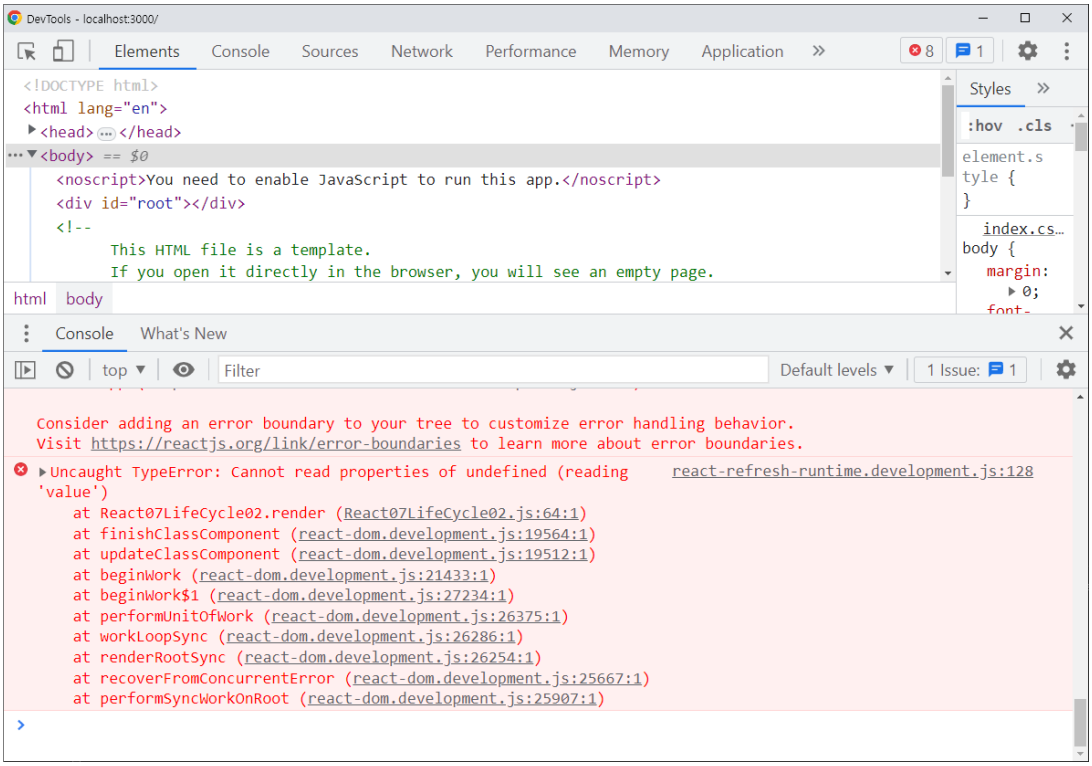
1. 에러발생시키기

```
src/App.js
import React, { Component } from 'react';
import React07LifeCycle01 from './mysrc/React07LifeCycle01';
import React07LifeCycle02 from './mysrc/React07LifeCycle02';

// 랜덤 색상을 생성합니다.
function getRandomColor() {
  return '#' + Math.floor(Math.random() * 16777215).toString(16);
}

class App extends Component {

  state = {
    color: '#000000'
  };
```

```
  handleClick = () => {
    this.setState({
      color: getRandomColor()
    });
  };

  render() {
    return (
      <div>
        <button onClick={this.handleClick}>랜덤 색상</button>
        {/* React07LifeCycle01에 color값을 props로 전달 */}
        <React07LifeCycle01 color={this.state.color} />
        <hr/>
        {/* 의도적인 에러 발생 */}
        <React07LifeCycle02 color={this.state.color} />
      </div>
    );
    }
}
export default App;

src/mysrc/React07LifeCycle02.js
import { Component } from 'react';

class React07LifeCycle02 extends Component {

  state = {
    number: 0,
    color: null,
  }

  myRef = null;

  constructor(props) {
    super(props);
    console.log('1. 생성자함수 호출!!')
  }

  static getDerivedStateFromProps(nextProps, prevState) {
    console.log("2.getDerivedStateFromProps메서드 호출!!");
    if(nextProps.color !== prevState.color) {
      return { color: nextProps.color };
    }
    return null;
  }

  componentDidMount() {
    console.log("3. componentDidMount 호출!! ");
  }

  shouldComponentUpdate(nextProps, nextState) {
    console.log("4. shouldComponentUpdate 호출!! ", nextProps,
nextState);
    return nextState.number % 10 !== 4;
  }
```

```
  componentWillUnmount() {
    console.log('5. componentWillUnmount 호출!!');
  }

  handleClick = () => {
    this.setState({
      number: this.state.number + 1,
    });
  };

  getSnapshotBeforeUpdate(prevProps, prevState) {
    console.log('6. getSnapshotBeforeUpdate 호출!!');
    if (prevProps.color !== this.props.color) {
      return this.myRef.style.color;
    }
    return null;
  }

  componentDidUpdate(prevProps, prevState, snapshot) {
    console.log('7. componentDidUpdate 호출!!', prevProps,
prevState);
    if (snapshot) {
      console.log('업데이트되기 직전 색상: ', snapshot);
    }
  }
  render() {
    console.log('8. render 호출!!');
    const style = {
      color: this.props.color,
    };
    return (
      <div>
        {/* 의도적인 에러 발생
            존재하지 않는 props인 missing객체를 조회해서 랜더링하면
            당연히 웹브라우저에서는 에러가 발생!!
        */}
        {this.props.missing.value}
        <h1 style={style} ref={(ref) => (this.myRef = ref)}>
          {this.state.number}
        </h1>
        <p>color: {this.state.color}</p>
        <button onClick={this.handleClick}>더하기</button>
      </div>
    );
  }
}

export default React07LifeCycle02;
```

- 에러가 발생하면 웹브라우저는 흰색화면만 출력된다. 이럴 때 사용자에게 인지 시켜줄 필요가 있다.



2. 에러처리하기

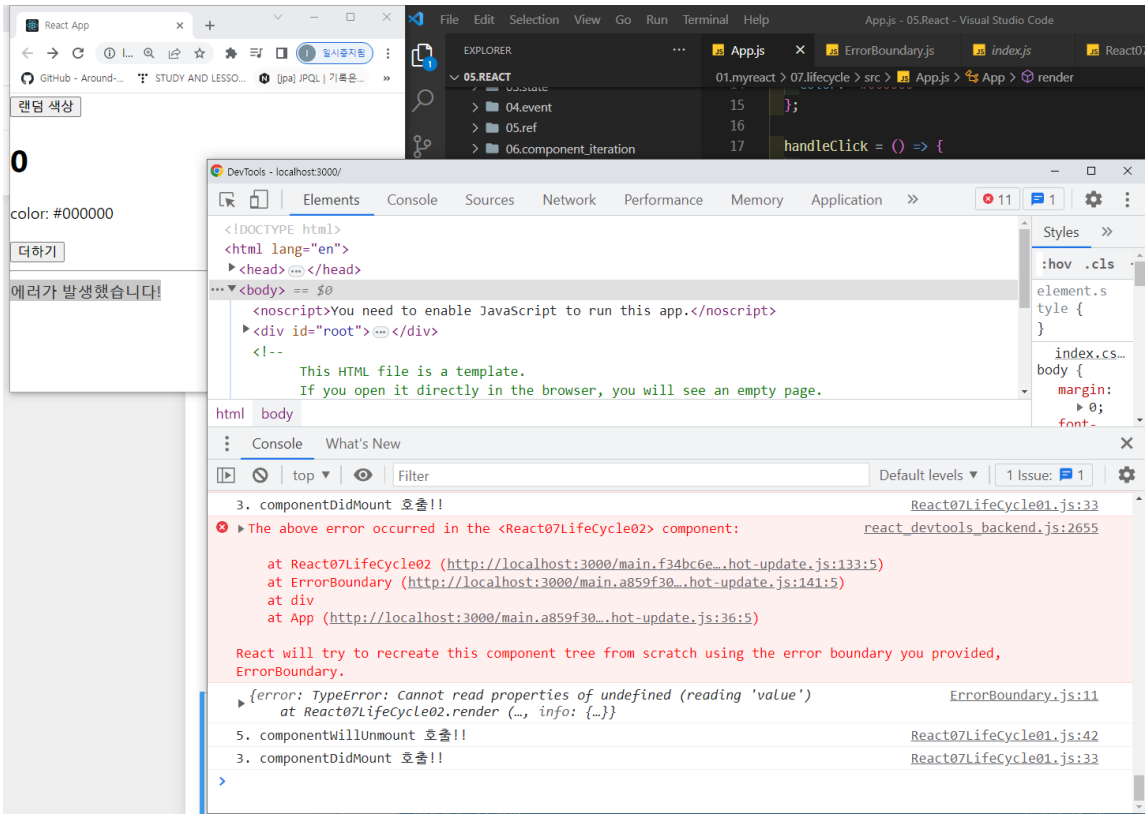
- 에러가 발생하면 componentDidCatch메서드가 호출되고 `this.state.error값을 true로 업데이트` 한다.

```
src/mysrc/ErrorBoundary.js
import React, { Component } from 'react';

class ErrorBoundary extends Component {
  state = {
    error: false
  };
  componentDidCatch(error, info) {
    this.setState({
      error: true
    });
    console.log({ error, info });
  }
  render() {
    if (this.state.error) return <div>에러가 발생했습니다!</div>;
    return this.props.children;
  }
}

export default ErrorBoundary;
```

``src/App.js import React, { Component } from 'react'; import ErrorBoundary from './mysrc/ErrorBoundary'; import React07LifeCycle01 from './mysrc/React07LifeCycle01'; import React07LifeCycle02 from './mysrc/React07LifeCycle02';

// 랜덤 색상을 생성합니다. function getRandomColor() { return '#' + Math.floor(Math.random() * 16777215).toString(16); }

```
class App extends Component {

state = { color: '#000000' };

handleClick = () => { this.setState({ color: getRandomColor() }); };

render() { return (

랜덤 색상 {/ 1. React07LifeCycle01에 color값을 props로 전달 /}

---

    {/* 2. 의도적인 에러 발생 */}
    {/* <React07LifeCycle02 color={this.state.color} /> */}

    {/* 3. 에러처리 */}
    <ErrorBoundary>
      <React07LifeCycle02 color={this.state.color} />
    </ErrorBoundary>
  </div>
);
}
} export default App; ```
```



In [ ]:

In [ ]: