섹션 4 리액트 심화, 틱택토

프래그먼트 사용법

JSX 문법은 상위 요소 하나를 반환해야한다

- 형제 요소들만 반환시 에러 발생
- 왜일까?
    - 반환값이 두개일 순 없다

```
return (
    <div>
        <Header />
        <main> ...
        </main>
    </div>
    )

    => 그러나 개발자도구 -> 불필요한 div가 생성된다
```

```
대안 !!

import { Fragment } from ‘react’

react 내장 Fragment 컴포넌트 사용

return (
    <Fragment>
        <Header />
```

```
        <main>
        </main>
    </Fragment>
    ))
```

```
더 짧게 사용도 가능

빈 태그 == fragment 컴포넌트
return (
    <>
        <Header />
        <main>
        </main>
    </>
    ))
```

컴포넌트 분리

App.jsx에 모든 컴포넌트를 다 때려넣으면 안된다

예를 들어,

CoreConcepts.jsx로 분리,

Examples.jsx로 분리

```
#examples.jsx

import { useState } from 'react'
import { EXAMPLES } from '../data.js'
import TabButton ..

export default function Examples() {
```

```
    const [selectedTopic, setSelectedTopic] = useState()
    function handleSelect(selectedButton) {
        setSelectedTopic(selectedButton)
    }

    let tabContent = <p> Please select a topic </p>

    if (selectedTopic) {
        tabContent = (
            <div id="tab-content">
                <h3> {EXAMPLES[selectedTopic].title}</h3>
            </div>
            )
        }

    return (
    <section id="examples"
    <h2> Examples </h2>
    <menu>
    <TabButton
        isSelected={selectedTopic === 'components'}
        onSelect={() => handleSelect('components')}
        >
            Components
        </TabButton>
        <TabButton>
            isSelected={selectedTopic === 'jsx'}
            onSelect={() => handleSelect('jsx')}
        > JSX
        </TabButton>

        )
```

#app.js

```
import Examples from './components/Examples.jsx'


function App() {
    return (
        <>
            <Header />
            <main>
            </main>
        </>
        ))
```

감싸진 요소에 props 전달하기

- 컴포넌트 분리할 때 ...props로, 간단하게 id값 등도 같이 보내줄 수 있다

```
#section.jsx

export default function Section({title, children, ...props}) {
    return (
        <section {...props}>
            <h2>{title}</h2>
                {children}
            </section>
        )
    }


#examples.jsx

return (
<Section title="Examples" id="examples" className=""> // ..props
```

```
    <menu>
    ...
```

```
...props 사용 시 주의사항

과도한 의존 방지
명확한 props가 필요할 때는 특정 props를 직접 선언하여 가독성 유지.
예: 필수적인 children이나 className은 명시적으로 정의.
...props 우선순위
컴포넌트 내부에서 직접 정의된 props가 ...props보다 우선 적용.
예시:내부에서 className을 재정의하면 className="primary"는 덮어써짐.
  <TabButton className="primary" data-custom="example" />
의미 없는 props 전달 방지
불필요한 props가 DOM 요소에 그대로 전달되지 않도록 주의.
```

슬롯(Slot)

- **슬롯**은 컴포넌트 외부에서 콘텐츠를 **동적으로 삽입**할 수 있는 영역.
- React에서는 `children` 프로퍼티나 특정 `props`를 사용하여 구현.
- 사용 이유
  - **구조와 콘텐츠 분리**: 컴포넌트 구조와 콘텐츠를 분리해 유지보수 용이.
  - **재사용성 증가**: 동일 컴포넌트에 다양한 콘텐츠 삽입 가능.
  - **유연성**: 외부 콘텐츠를 동적으로 삽입하여 컴포넌트 구성 변경 가능.

여러 JSX 활용법

JSX와 Props를 통한 슬롯

- **JSX**는 값으로 취급되어 **props로 전달** 후 컴포넌트 내부에서 렌더링.
- **주의사항** : jsx코드를 하나의 값으로 사용할 때는 오직 하나의 루트 요소만 허용되므로, 최상위 요소 하나가 있어야 함

예시:

```
// Tabs.jsx
export default function Tabs({ children, buttons }) {
  return (
    <>
      <menu>{buttons}</menu>  {/* 버튼 슬롯 */}
      {children}  {/* 메인 콘텐츠 슬롯 */}
    </>
  );
}

// Examples.jsx
export default function Examples() {
  return (
    <Tabs
      buttons={
        <>
          <TabButton>Components</TabButton>
          <TabButton>JSX</TabButton>
        </>
      }
    >
      <div>Selected Content</div> {/* 메인 콘텐츠 슬롯 */}
    </Tabs>
  );
}
```

컴포넌트 타입 동적으로 사용하기

tabs.jsx

```jsx
// 속성 추가
function Tabs({children, buttons, buttonsContainer }) {
    const ButtonContainer = buttonsContainer
    // 변수 생성
    // 아니면 애초에 속성값 추가할때 대문자로 추가하면 됨
    return (
        <>
            <ButtonContainer>{buttons}</ButtonContainer>
                {children}
        </>
    )
}
```

examples.jsx

```jsx
return (
    <Section title="Examples" id="examples">
        <Tabs
            buttonsContainer="ul" //ul, div, menu, {컴포넌트} 등 D
            // 내장 컴포넌트는 문자열로 ㅇㅇ
            // 커스텀 컴포넌트는 {컴포넌트이름}
            buttons={
                <>
                ..
```

기본 props 값 설정

```
tabs.jsx

// 속성 default 설정하기
function Tabs({children, buttons, ButtonsContainer ='menu' }) {

    return (
        <>
            <ButtonContainer>{buttons}</ButtonContainer>
                {children}
        </>
    )
}
```

이미지 저장소는 public/ VS assets/

public/ 폴더

이전 강의에서 보았듯이, 이미지를 `public/` 폴더에 저장하고 `index.html` 또는 `index.css` 파일 내에서 **직접 참조**할 수 있습니다.

이렇게 하는 이유는 `public/`에 저장된 이미지 (또는 일반적으로: 파일)이 프로젝트 개발 서버 및 빌드 프로세스에 의해 **공개적으로 제공**되기 때문입니다. `index.html`과 마찬가지로, 이 파일들은 브라우저 내에서 직접 방문할 수 있으며, 따라서 다른 파일에 의해 요청될 수도 있습니다.

예를 들어, `localhost:5173/some-image.jpg`를 불러오면 해당 이미지를 볼 수 있습니다 (물론 `public/` 폴더에 이미지가 있을 경우).

src/assets/ 폴더

이미지를 `src/assets/` 폴더 (또는 실제로는 `src` 폴더의 어디든)에 저장할 수도 있습니다.

그렇다면 `public/`와 비교해 어떤 차이가 있을까요?

`src` 또는 `src/assets/`와 같은 하위 폴더에 저장된 모든 파일(어떤 형식이든)은 공개적으로 제공되지 않습니다. 웹사이트 방문자가 접근할 수 없습니다. `localhost:5173/src/assets/some-`

`image.jpg` 를 불러오려고 하면 오류가 발생합니다.

대신, `src/` (및 하위 폴더)에 저장된 파일은 코드 파일에서 사용할 수 있습니다. 코드 파일에 가져온 이미지는 빌드 프로세스에 의해 인식되어 최적화되며, 웹사이트에 제공하기 직전에 `public/` 폴더에 "삽입"됩니다. 가져온 이미지는 참조한 위치에서 자동으로 링크가 생성되어 사용됩니다.

어떤 폴더를 사용해야 할까요?

빌드 프로세스에 의해 처리되지 않는 이미지는 `public/` 폴더를 사용해야 하고 **대체적으로 사용 가능** 가능합니다. 예를 들면 `index.html` 파일이나 파비콘과 같은 이미지가 좋은 후보입니다.

반면, **컴포넌트 내**에서 사용되는 이미지는 일반적으로 `src/` 폴더(예: `src/assets/`)에 저장되어야 합니다.

실습 : 틱택토 게임

- 컴포넌트 분리

```
Players.jsx

// props 속성 전달
export default function Player({ name, symbol }) {
  return (
    <li>
      <span className="player">
        <span className="player-name">{name}</span>
        <span className="player-symbol">{symbol}</span>
      </span>
      <button>Edit</button>
    </li>
  );
}
```

App.jsx

```jsx
<div id="game-container">
  <ol id="players">
    <Player name="Player 1" symbol="X" />
    <Player name="Player 2" symbol="0" />
  </ol>
</div>
```

- state 관리

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

export default function Player({ name, symbol }) {
  // 사용자정보 수정 state 정의
  // 해당 컴포넌트 변화 있는지 재평가
```

```
  const [isEditing, setIsEditing] = useState(false);
  // isEditing : Ui 요소 조정
  // setIsEditing : isEditing 조정 변수


  function handleEditClick() {
    setIsEditing(!isEditing); // true 로 변환 // toggle 기능이라 보
    // ! 하나로 토글 기능 구현 끝
  }

  //jsx코드를 하나의 변수로 정의 가능
  let playerName = <span className="player-name">{name}</span>;

  if (isEditing) {
    playerName = <input type="text" required />;
  }
  return (
    <li>
      <span className="player">
        {playerName}
        <span className="player-symbol">{symbol}</span>
      </span>
      {/* 함수 전달위해 함수 이름만 작성 */}
      <button onClick={handleEditClick}>{isEditing ? 'Save': 'Ed
    </li>
  );
}
```

```
ㅇ; // true 로 변환 // toggle 기능이라 보면 될듯
    // ! 하나로 토글 기능 구현 끝
  }
```

주의 2

```
  // value = 기존 입력 value
  // defaultValue로 설정시 특정값 계속 덮어쓰기 막아줌
  if (isEditing) {
    playerName = <input type="text" required defaultValue={name}
  }
```

- onChange와 사용자입력(수정)값 상태 참조 == 양방향 바인딩

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

// 인자를 name에서 이니셜 name을 변경(명시적)
export default function Player({ initialName, symbol }) {
  // 사용자이름 edit할 경우 track할(상태관리) 변수가 필요
  const [playerName, setPlayerName] = useState(initialName);

  // 사용자정보 수정 state 정의
  // 해당 컴포넌트 변화 있는지 재평가
  const [isEditing, setIsEditing] = useState(false);

  function handleEditClick() {
    setIsEditing((editing) => !editing); // true 로 변환 // toggl
    // ! 하나로 토글 기능 구현 끝
  }

  // 사용자 입력(수정) 이름 반영 처리할 함수 생성
  function handleChange(event) {
    console.log(event);
    setPlayerName(event.target.value); //사용자 입력 받기
  }
  //jsx코드를 하나의 변수로 정의 가능
  let editablePlayerName = <span className="player-name">{playe

  // value = 기존 입력 value
  // defaulyValue로 설정시 특정값 계속 덮어쓰기 막아줌
```

```
  if (isEditing) {
    editablePlayerName = (
      <input
        type="text"
        required
        defaultValue={playerName} // 변수값 표시 {}
        onChange={handleChange}
      />
    );
  }
  return (
    <li>
      <span className="player">
        {editablePlayerName}
        <span className="player-symbol">{symbol}</span>
      </span>
      {/* 함수 전달위해 함수 이름만 작성 */}
      <button onClick={handleEditClick}>{isEditing ? "Save" : "E
    </li>
  );
}
```

실습 틱택토 최종 코드 🔥

App.jsx

state ⇒ prev

https://1yoouoo.tistory.com/16

```
// player.js와 gameboard.js의 가장 가까운 부모 컴포넌트에서
// 어느 플레이어가 플레이를 하고 있는지 상태 추적
```

```jsx
import { useState } from "react";
import Player from "./components/Player.jsx";
import GameBoard from "./components/GameBoard.jsx";
import Log from "./components/Log.jsx";
import { WINNING_COMBINATIONS } from "./winning-combinations.js
import GameOver from "./components/GameOver.jsx";

// 이 앱의 상수 정의 (대문자)
const PLAYERS = {
  X: "Player 1",
  O: "Player 2",
};

// 3x3 격자판을 나타내는 초기 게임 보드 상태
const INITIAL_GAME_BOARD = [
  [null, null, null],
  [null, null, null],
  [null, null, null],
];

function deriveGameBoard(gameTurns) {
  // driving state (GameBoard computed value) !!
  // deep copy(안의 값이 함께 변경되지 않게)
  // let gameBoard = initialGameBoard 가 아닌 아래처럼 해야함
  let gameBoard = [...INITIAL_GAME_BOARD.map((array) => [...arra

  for (const turn of gameTurns) {
    const { square, player } = turn;
    const { row, col } = square;

    gameBoard[row][col] = player;
  }
  return gameBoard;
}

// 많은 STATE 관리는 좋지 않다, 간결성 필요
```

```
function deriveActivePlayer(gameTurns) {
  let currentPlayer = "X";

  if (gameTurns.length > 0 && gameTurns[0].player === "X") {
    currentPlayer = "O";
  }
  return currentPlayer;
}

function deriveWinner(gameBoard, players) {
  // 게임종료조건설정, 우승자 명시
  let winner = null;

  for (const combination of WINNING_COMBINATIONS) {
    // [0].row란 가로 한줄
    const firstSquareSymbol =
      gameBoard[combination[0].row][combination[0].column];
    const secondSquareSymbol =
      gameBoard[combination[1].row][combination[1].column];
    const thirdSquareSymbol =
      gameBoard[combination[2].row][combination[2].column];

    // null은 false로 여겨짐
    if (
      firstSquareSymbol &&
      firstSquareSymbol === secondSquareSymbol &&
      firstSquareSymbol === thirdSquareSymbol
    ) {
      winner = players[firstSquareSymbol];
    }
  }
  return winner;
}

function App() {
  const [players, setPlayers] = useState(PLAYERS);
```

```
  // log 기록을 위해 플레이어 순서 정보 저장
  const [gameTurns, setGameTurns] = useState([]);

  // state 파생
  const activePlayer = deriveActivePlayer(gameTurns);
  const gameBoard = deriveGameBoard(gameTurns);

  const winner = deriveWinner(gameBoard, players);

  //무승부 판정 변수
  const hasDraw = gameTurns.length === 9 && !winner;

  //칸 선택할때마다 플레이어 전환
  function handleSelectSquare(rowIndex, colIndex) {
    // setActivePlayer((curActivePlayer) => (curActivePlayer ===

    // prev :
    // log 기록을 위해 플레이어 순서 업데이트
    setGameTurns((prevTurns) => {
      const currentPlayer = deriveActivePlayer(prevTurns);

      //이전 turn 앞쪽에 쌓이는 구조
      const updatedTurns = [
        { square: { row: rowIndex, col: colIndex }, player: acti
        ...prevTurns,
      ];
      return updatedTurns;
    });
  }

  function handleRestart() {
    setGameTurns([]); // turn 초기화
  }

  function handlePlayerNameChange(symbol, newName) {
    setPlayers((prevPlayers) => {
```

```
      return {
        ...prevPlayers,
        [symbol]: newName,
      };
    });
  }

  return (
    <main>
      <div id="game-container">
        <ol id="players" className="highlight-player">
          {/* 현재 플레이어 동적으로 표시 */}
          <Player
            initialName={PLAYERS.X}
            symbol="X"
            // props로 전달
            isActive={activePlayer === "X"}
            onChangeName={handlePlayerNameChange}
          />
          <Player
            initialName={PLAYERS.O}
            symbol="O"
            isActive={activePlayer === "O"}
            onChangeName={handlePlayerNameChange}
          />
        </ol>
        {(winner || hasDraw) && (
          <GameOver winner={winner} onRestart={handleRestart} />
        )}
        <GameBoard onSelectSquare={handleSelectSquare} board={ga
        {/* activePlayer라는 props 전달 */}
        {/*격자판 컴포넌트 */}
      </div>
      <Log turns={gameTurns} />
    </main>
  );
```

```
}

export default App;
```

Player.jsx

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

// 인자를 name에서 이니셜 name을 변경(명시적)
export default function Player({
  initialName,
  symbol,
  isActive,
  onChangeName,
}) {
  // 사용자이름 edit할 경우 track할(상태관리) 변수가 필요
  const [playerName, setPlayerName] = useState(initialName);

  // 사용자정보 수정 state 정의
  // 해당 컴포넌트 변화 있는지 재평가
  const [isEditing, setIsEditing] = useState(false);

  function handleEditClick() {
    setIsEditing((editing) => !editing); // true 로 변환 // toggl
    // ! 하나로 토글 기능 구현 끝

    if (isEditing) {
      onChangeName(symbol, playerName);
    }
  }

  // 사용자 입력(수정) 이름 반영 처리할 함수
  function handleChange(event) {
    console.log("event", event);
    setPlayerName(event.target.value); //사용자 입력 받기
  }
```

```
  //jsx코드를 하나의 변수로 정의 가능
  let editablePlayerName = <span className="player-name">{playe

  // value = 기존 입력 value
  // defaulyValue로 설정시 특정값 계속 덮어쓰기 막아줌
  if (isEditing) {
    editablePlayerName = (
      <input
        type="text"
        required
        defaultValue={playerName}
        onChange={handleChange}
      />
    );
  }
  return (
    <li className={isActive ? "active" : undefined}>
      <span className="player">
        {editablePlayerName}
        <span className="player-symbol">{symbol}</span>
      </span>
      {/* 함수 전달위해 함수 이름만 작성 */}
      <button onClick={handleEditClick}>{isEditing ? "Save" : "E
    </li>
  );
}
```

GameOver.jsx

```
// 무승부일땐 winner가 null
export default function GameOver({ winner, onRestart }) {
  return (
    <div id="game-over">
      <h2>Game over</h2>
      {winner && <p> {winner} won!</p>}
      {!winner && <p> it's a draw</p>}
```

```
      <button onClick={onRestart}>rematch!</button>
    </div>
  );
}
```

GameBoard.jsx

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

// gameboard 자식 컴포넌트는 부모로부터 onSelectSquare props 전달받음
// 부모 컴포넌트 상태 변경도 가능 !!
export default function GameBoard({ onSelectSquare, board }) {
  // // driving state (GameBoard computed value) !!
  // let gameBoard = initialGameBoard;

  // for (const turn of turns) {
  //   const { square, player } = turn;
  //   const { row, col } = square;

  //   gameBoard[row][col] = player;
  // }
  // const [gameBoard, setGameBoard] = useState(initialGameBoard

  // // 행, 렬 인덱스 전달
  // // 상태가 객체 혹은 배열일때 복제해서 상태 추적하는 게 좋음
  // // 메모리 속의 기존값 변경 == 비동기적 처리라 더 이전에 업데이트될 수
  // function handleSelectSquare(rowIndex, colIndex) {
  //   setGameBoard((previousGameBoard) => {
  //     const updateGameBoard = [
  //       ...previousGameBoard.map((innerArray) => [...innerAr
  //     ]; // 복제(깊은복사)

  //     updateGameBoard[rowIndex][colIndex] = activePlayerSymbo
  //     return updateGameBoard; // x,y 값 설정
  //   }); // 직전 상태 기반하여 업데이트
```

```
  //   // 게임보드 상태 업데이트 후 사용자 전환 함수
  //   onSelectSquare(); //버튼 클릭 시점에 작동 //부모의 handleSelec
  // }

  //랜더링 결과
  // 상태가(useState으로 추적하는) 변경될 때마다 다시 실행
  return (
    <ol id="game-board">
      {board.map((row, rowIndex) => (
        // 각 행을 리스트 아이템으로 렌더링
        <li key={rowIndex}>
          <ol>
            {row.map((playerSymbol, colIndex) => (
              // 각 열을 리스트 아이템으로 렌더링
              <li key={colIndex}>
                {/* 인자를 함수로, full control */}
                <button
                  onClick={() => onSelectSquare(rowIndex, colInd
                  // 동기 처리 (버튼 활성화 - 한곳에 한번만 클릭되게)
                  // 플레이터 턴 표시
                  // x, o 일때 true
                  disabled={playerSymbol !== null}
                >
                  {playerSymbol}
                </button>
              </li> // 이벤트 핸들링  함수 직접 전달 : 지연실행 !!
            ))}
          </ol>
        </li>
      ))}
    </ol>
  );
}
```

Log.jsx

```jsx
// Log.jsx : 플레이마다 어떤 수 뒀는지 로그 컴포넌트

export default function Log({ turns }) {
  return (
    <ol id="log">
      {turns.map((turn) => (
        <li key={`${turn.square.row}${turn.square.col}`}>
          {turn.player} selected {turn.square.row}, {turn.square
        </li>
      ))}
    </ol>
  );
}
```