



8장-타입스크립트로 리액트 컴포넌트 만들기 (feat. Select 컴포넌트)

💠 TS로 리액트 컴포넌트 만들기

8.2.1 JSX로 구현된 Select 컴포넌트

추가 설명이 없다면 컴포넌트를 사용하는 입장에서 각 속성에 어떤 타입의 값을 전달해야 할지 알기 어렵다.

```
// onChange : 부모 컴포넌트에서 전달받은 콜백함수
const Select = ({ onChange, options, selectedOption }) => {

  const handleChange = (e) => {
    const selected = Object.entries(options).find(
      ([_, value]) => value === e.target.value // 선택된 값과 일치하
    )?.[0]; // undefined일 경우 예러 방지
    // 부모로부터 받은 onChange 함수 호출하며 찾은 key를 전달
    onChange?.(selected);
  };

  return (
    <select
      onChange={handleChange}
      value={selectedOption && options[selectedOption]}
    >
      {Object.entries(options).map(([key, value]) => (
        <option key={key} value={value}>
          {value}
        </option>
      ))}
    </select>
  );
};
```



▼ `Object.entries()`

: 객체를 [key, value] 쌍의 배열로 변환한다.

```
const obj = {
  name: "김철수",
  age: 25
};

const entries = Object.entries(obj);
console.log(entries);
// 출력: [["name", "김철수"], ["age", 25]]
```



▼ `find()`

: 배열에서 조건을 만족하는 **첫 번째 요소**를 반환한다.

- 조건은 콜백함수로 지정한다.
- 콜백함수의 매개변수인 `element`, `index`, `array`는 선택 사용

```
array.find((element, index, array) => {
  // element: 배열에서 현재 처리 중인 요소
  // index: 현재 처리 중인 요소의 인덱스
  // array: find()를 호출한 배열 자체

  // 조건식 return
});
```

✓ `find()` vs. `filter()`

Feature	<code>find()</code>	<code>filter()</code>
반환값	첫 번째 요소	조건을 만족하는 모든 요소를 배열로 반환
타입	단일 요소 또는 <code>undefined</code>	배열
순회 중단	조건을 만족하면 중단	끝까지 순회
사용 예시	조건 만족하는 첫 번째 요소 찾기	조건 만족하는 여러 요소 찾기

8.2.2 JSDocs로 일부 타입 지정하기


컴포넌트의 속성 타입을 명시하는 방법 : **JSDocs**

컴포넌트에 대한 설명과 각 속성이 어떤 역할을 하는지 간단하게 알려준다.

```
/**
 * Select 컴포넌트
 * @param {Object} props - Select 컴포넌트로 넘겨주는 속성
 * @param {Object} props.options - { [key: string]: string } 형식의 객체
 * @param {string | undefined} props.selectedOption - 현재 선택된 옵션
 * @param {function} props.onChange - select 값이 변경되었을 때 불리는 함수
 * @returns {JSX.Element}
 */
const Select = (...
```

8.2.3 props 인터페이스 적용하기

! JSDocs는 options가 어떤 형식의 객체를 나타내는지, onChange의 매개변수 및 반환값에 대한 구체적인 정보를 알기 쉽지 않아서 잘못된 타입이 전달될 수 있다.

 TS를 통해 구체적인 타입을 지정

JSX 파일 → TSX 파일

Select 컴포넌트의 props에 대한 인터페이스

✓ TS로 인해 알 수 있는 사실

- options 타입 정의 : string만
- onChange : 선택된 string값을 매개변수로 받고 어떤 값도 반환하지 않는 함수

```
type Option = Record<string, string>; // {[key: string]: string}

interface SelectProps {
  options: Option;
  selectedOption?: string;
  onChange?: (selected?: string) => void;
}

const Select = ({
```

```

options,
selectedOption,
onChange,
}: SelectProps): JSX.Element => {
  //...
};

```

8.2.4 리액트 이벤트

리액트는 가상 DOM을 다루면서 이벤트도 별도로 관리한다.

- 이벤트 리스너의 네이밍 규칙
 - 일반 DOM: `onclick`, `onchange` (소문자)
 - 리액트: `onClick`, `onChange` (카멜케이스)

→ 리액트 이벤트는 브라우저의 고유한 이벤트와 동일하기 동작 X

- 이벤트 핸들러는 이벤트 버블링 단계에서 호출된다.

이벤트 버블링 : 이벤트가 실제 타깃 요소에서 시작해서 상위 요소로 올라가는 단계 (아래에서 위로)

- 이벤트 캡처 단계에서 이벤트 핸들러를 등록하기 위해서는 `onClickCapture`, `onChangeCapture`와 같이 일반 이벤트 리스너 이름 뒤에 `Capture`를 붙여야 한다.

이벤트 캡처링 : 이벤트가 최상위 요소에서 시작해서 실제 타깃 요소까지 내려가는 단계 (위에서 아래로)

- 브라우저 이벤트를 합성한 합성 이벤트를 제공한다. `SyntheticEvent`

합성 이벤트 : 브라우저마다 다르게 동작하는 이벤트들을 리액트가 한번 감싸서 통일된 방식으로 동작하게 만든 것

```

// 리액트의 이벤트 핸들러 타입 정의
type EventHandler<Event extends React.SyntheticEvent> = (
  e: Event
) => void | null;
// select 엘리먼트의 change 이벤트를 위한 구체적인 타입

```

```

type ChangeEventHandler = EventHandler<ChangeEvent<HTMLSelectElement>
// changeEvent<HTMLSelectElement> : 리액트의 합성이벤트 중 onChange

const eventHandler1: GlobalEventHandlers["onChange"] = (e) => {
  e.target; // 일반 Event는 target이 없음
};

const eventHandler2: ChangeEventHandler = (e) => {
  e.target; // 리액트 이벤트(합성 이벤트)는 target이 있음
};

```

- `React.ChangeEventHandler<HTMLSelectElement>` = `React.EventHandler<ChangeEvent<HTMLSelectElement>>` 타입

```

type ChangeEventHandler<T> = EventHandler<ChangeEvent<T>>;

```

```

const Select = ({ onChange, options, selectedOption }: SelectProps) => {
  // const handleChange = (e) => {
  const handleChange: React.ChangeEventHandler<HTMLSelectElement> = (e) => {
    const selected = Object.entries(options).find(
      ([_, value]) => value === e.target.value
    )?.[0];
    onChange?.(selected);
  };

  return <select onChange={handleChange}>{/* ... */</select>;
};

```

8.2.5 후에 타입 추가하기

Select 컴포넌트를 사용하여 과일을 선택할 수 있는 컴포넌트

```

const fruits = { apple: "사과", banana: "바나나", blueberry: "블루베리" };

const FruitSelect: VFC = () => {
  const [fruit, changeFruit] = useState<string | undefined>();

```

```

    return (
      <Select
        onChange={changeFruit}
        options={fruits}
        selectedOption={fruit}
      />
    );
  };
};

```

❗ 타입 매개변수가 없다면, fruit의 타입이 undefined로만 추론되어 onChange의 타입과 일치 않아 오류 발생

❗ 제네릭 타입을 지정 X으면, 작성자는 fruit가 반드시 apple, banana, blueberry 중 하나라고 기대하지만 → TS 컴파일러는 fruit를 string으로 추론하고, 다른 개발자가 changeFruit에 orange를 넣을 수도 있다.

사이드 이펙트 : 프로그램의 실행 결과가 예상치 못한 상태로 변경되거나 예상치 못한 동작을 하게 되는 상황을 가리킨다. 즉, 코드의 실행이 예상과 다르게 동작하여 예상치 못한 결과를 초래하는 것

```

type Fruit = keyof typeof fruits; // 'apple' | 'banana' | 'blueberry'
const [fruit, changeFruit] = useState<Fruit | undefined>("apple")

// 에러 발생
const func = () => {
  changeFruit("orange");
};

```

8.2.6 제네릭 컴포넌트 만들기

Select 컴포넌트를 사용하는 입장에서 제한된 key, value만을 가지게 하고 싶다. → 제네릭을 사용한 컴포넌트

▼ before

```

type Option = Record<string, string>; // {[key: string]: string}

interface SelectProps {
  options: Option;
  selectedOption?: string;
  onChange?: (selected?: string) => void;
}

const Select = ({
  options,
  selectedOption,
  onChange,
}: SelectProps): JSX.Element => {
  //...
};

```

- <Select<추론된_타입>> 형태의 컴포넌트

```

interface SelectProps<OptionType extends Record<string, string>> {
  options: OptionType;
  selectedOption?: keyof OptionType;
  onChange?: (selected?: keyof OptionType) => void;
}

const Select = <OptionType extends Record<string, string>>({
  options,
  selectedOption,
  onChange,
}: SelectProps<OptionType>) => {
  // Select component implementation
};

```

8.2.7 HTMLAttributes, ReactProps 적용하기

SelectProps에 직접 넣어도 되지만, 리액트 제공 타입을 사용하면 더 정확한 타입 설정이 가능하다.

```
type ReactSelectProps = React.ComponentPropsWithoutRef<"select">

interface SelectProps<OptionType extends Record<string, string>>
  id?: ReactSelectProps["id"];
  className?: ReactSelectProps["className"];
  // ...
}
```


- `ComponentPropsWithoutRef` : 리액트 컴포넌트의 prop 타입을 반환해주는 타입
- HTML select 엘리먼트의 기본 속성들을 그대로 사용할 수 있으면서, 커스텀 props도 추가할 수 있다.

ReactProps에서 여러 개의 타입을 가져와야 한다면 `Pick` 키워드를 활용한다.

```
interface SelectProps<OptionType extends Record<string, string>>
  extends Pick<ReactSelectProps, "id" | "key" | /* ... */> {
  // ...
}
```

8.2.8 styled-components를 활용한 스타일 정의

`CSS-in-JS` : CSS 파일 대신 JS 안에 직접 스타일을 정의하는 기법

 Select 컴포넌트에 `fontSize`와 현재 선택된 option의 글꼴 색상을 설정해보자

1. theme 객체를 생성하고, 프로젝트에서 사용될 `fontSize`, `color`, 해당 타입을 구성한다.

```
const theme = {
  fontSize: {
    default: "16px",
    small: "14px",
    large: "18px",
  },
  color: {
    white: "#FFFFFF",
    black: "#000000",
  },
}
```



```
};
```

```
type Theme = typeof theme;  
type FontSize = keyof Theme["fontSize"];  
type Color = keyof Theme["color"];
```

```
const theme: {  
  fontSize: {  
    default: string;  
    small: string;  
    large: string;  
  };  
  color: {  
    white: string;  
    black: string;  
  };  
}
```

```
(property) fontSize: {  
  default: string;  
  small: string;  
  large: string;  
}
```

```
(property) color: {  
  white: string;  
  black: string;  
}
```

2. 스타일과 관련된 props를 작성하고, color와 font-size 스타일 정의를 담은 StyledSelect를 작성한다.

▼ 실제 사용

```
// 2. 실제 사용  
return (  
  // select 태그  
  <StyledSelect  
    color="primary"  
    fontSize="md"  
    onChange={handleChange}  
  >  
    <option value="1">옵션 1</option>  
    <option value="2">옵션 2</option>  
  </StyledSelect>  
)
```

```
interface SelectStyleProps {
  color: Color;
  fontSize: FontSize;
}

const StyledSelect = styled.select<SelectStyleProps>`
  color: ${({ color }) => theme.color[color]};
  font-size: ${({ fontSize }) => theme.fontSize[fontSize]};
`;
```

3. Select를 사용하는 부모 컴포넌트에서 원하는 스타일을 적용하기 위해, Select 컴포넌트의 props에 `SelectStyleProps` 타입을 상속한다.

- `Partial<Type>` : 객체 형식의 타입 내 모든 속성을 옵셔널로 설정

```
interface SelectProps extends Partial<SelectStyleProps> {
  // ...
}

const Select = <OptionType extends Record<string, string>>({
  fontSize = "default",
  color = "black",
}): // ...
SelectProps<OptionType>) => {
  // ...

  return (
    <StyledSelect
      // ...
      fontSize={fontSize}
      color={color}
      // ...
    />
  );
};
```

▼ 전체 코드

```

// 1. 스타일 props 정의
interface SelectStyleProps {
  color: Color;
  fontSize: FontSize;
}

// 2. Select 컴포넌트의 props 정의
interface SelectProps<OptionType extends Record<string, string>> {
  onChange?: (selected?: keyof OptionType) => void;
  options: OptionType;
  selectedOption?: keyof OptionType;
  // ... 다른 props
}

// 3. Select 컴포넌트
const Select = <OptionType extends Record<string, string>>({
  fontSize = "default", // 기본값 설정
  color = "black",      // 기본값 설정
  onChange,
  options,
  selectedOption,
  // ... 다른 props
}: SelectProps<OptionType>) => {
  return (
    <StyledSelect
      fontSize={fontSize}
      color={color}
      onChange={handleChange}
      value={selectedOption && options[selectedOption]}
    >
      {/* ... options 렌더링 */}
    </StyledSelect>
  );
};

```

? extends는 제한하는 것이니까 SelectProps에 SelectStyleProps 형식만 가능하다는
미인가?

```
interface SelectProps<OptionType> extends Record<string, string>
```

✓ extends 활용

- 타입을 확장할 경우 interface
- 타입을 조건부로 설정할 경우
- 제네릭 타입에서는 한정자 역할로 사용된다.

SelectStyleProps의 속성들을 포함하면서 다른 속성들도 추가할 수 있다는 의미이다.

8.2.9 공변성과 반공변성



용어 정리

넓은 타입 : 부모

좁은 타입 : 자식

$A = B$

: B를 A에 할당할 수 있다.

: A에서 B로 할당 가능하다.

Member : User보다 더 좁은 타입이자 User의 서브타입

공변성

타입 B

Member가 A User의 서브타입일 때, $T < B >$ 가 $T < A >$ 의 서브타입이 된다 =

공변성을 띠고 있다

(부모A = 자식B)

```
// 모든 유저(회원, 비회원)은 id를 갖고 있음
interface User {
  id: string;
}

interface Member extends User {
  nickName: string;
}

let users: Array<User> = [];
let members: Array<Member> = [];

users = members; // ✅ OK
members = users; // ❌ Error
```

🌟 일반적인 타입들은 **공변성**을 가지고 있어서, **넓은 타입** `users`에서 **좁은 타입** `members`으로 할당이 가능하다. BUT 제네릭 타입을 지닌 함수는 **반공변성**을 가진다.

반공변성

$T < B$

자식가 $T < A$ **부모**의 서브타입이 되어, 좁은 타입 $T < B$ 의 함수를 넓은 타입 $T < A$ 의 함수에 적용할 수 X다.

(부모A ≠ 자식B)

```
type PrintUserInfo<U extends User> = (user: U) => void;

// User는 id 프로퍼티를 지닌다.
let printUser: PrintUserInfo<User> = (user) => console.log(user.id);

// Member는 id, nickname 프로퍼티를 지닌다.
let printMember: PrintUserInfo<Member> = (user) => {
  console.log(user.id, user.nickName); // nickName까지 출력
};

printMember = printUser; // ✅ OK.
printUser = printMember; // ❌ Error - Property 'nickName' is missing in type 'User' but required in type 'Member'.
```

printUser는 PrintUserInfo<User> 타입이기 때문에 Member 타입을 매개변수로 받을 수 없다. → printMember 함수를 printUser 변수에 할당할 수 X다.

? 근데 PrintUserInfo의 제네릭에 `extends User` 라고 되어있는데, Member 타입을 넘겨줘도 되는걸까?

O

Member가 User의 모든 속성을 포함하고 있기 때문이다!

```
interface User {
  id: string;
}

interface Member extends User {
  nickName: string;
}

// User = Member OK
// Member = User Error
```

✓ 객체의 메서드 타입을 정의하는 방법 2가지

🌟 `--strict모드` 에서 onChangeA같이 함수 타입을 화살표 표기법으로 작성하면, `반공변성` 을 띠게 된다.

🌟 onChangeB와 같이 함수 타입을 지정하면, 공변성과 반공변성을 모두 가지는 `이변성` 을 띠게 된다.

→ 안전한 타입 가드를 위해 일반적으로 반공변적인 함수 타입을 설정하는 것이 권장된다.

```
interface Props<T extends string> {
  onChangeA?: (selected: T) => void; // 방법 1 : 반공변성
  onChangeB?(selected: T): void; // 방법 2 : 공변성, 반공변성(이변성)
}

const Component = () => {
  // 매개변수가 apple일 때 실행되는 메서드
  const changeToPineApple = (selectedApple: "apple") => {
    console.log("this is pine" + selectedApple);
  };
}
```

```

};

return (
  <Select
    // 🚨 Error
    // onChangeA={changeToPineApple}

    // ✅ OK
    onChangeB={changeToPineApple}
  />
);
};

```

! onChangeA는 **반공변성** 만 가지므로 더 구체적인 타입 **apple** 을 일반적인 타입 **string** 에 할당할 수 X다!

- onChangeA는 반공변성만 가지므로 타입이 더 엄격하게 검사된다.

```

type T extends string // onChangeA 타입: 일반적 - 부모(넓은 타입)
type "apple" // changeToPineApple 타입: 구체적 - 자식(좁은 타입)

```



일반적인 타입들은 넓은 타입 **부모** 에서 좁은 타입 **자식** 으로 할당 가능하다 → **공변성**
A=B

(= 좁은 타입 **자식** 을 넓은 타입 **부모** 에 할당할 수 있다)

제네릭 타입을 지닌 함수는 좁은 타입 **자식** 에서 넓은 타입 **부모** 으로 할당 가능하다 → **반공변성** B=A

- —strict 모드의 함수 타입
 - 화살표 함수 형식 - **반공변성** (**권장**)
 - 메서드 형식 - **이변성** 공변성/반공변성



그러면 객체의 메서드 타입을 정의할 때 화살표 함수 형식만 사용하는게 좋을까?

실제 개발에서는:

- 특별한 이유가 없다면 메서드 형식을 사용하는 것이 일반적
- 매우 엄격한 타입 체크가 필요한 경우에만 화살표 함수 형식 사용

▼ 잘못된? 책 설명

8.2.9 공변성과 반공변성

Member : User보다 더 좁은 타입이자 User의 서브타입

공변성

타입 A

Member가 B User의 서브타입일 때, T<A>가 T의 서브타입이 된

다 = 공변성을 띠고 있다

```
// 모든 유저(회원, 비회원)은 id를 갖고 있음
interface User {
  id: string;
}

interface Member extends User {
  nickName: string;
}

let users: Array<User> = [];
let members: Array<Member> = [];

users = members; // ✅ OK
members = users; // ❌ Error
```

🌟 일반적인 타입들은 공변성을 가지고 있어서, 좁은 타입 users에서 넓은 타입 members으로 할당이 가능하다. BUT 제네릭 타입을 지닌 함수는 반공변성을 가진다.

반공변성

T가 T<A>의 서브타입이 되어, 좁은 타입 T<A>의 함수를 넓은

타입 T의 함수에 적용할 수 X다.

```
type PrintUserInfo<U extends User> = (user: U) => void;

// User는 id 프로퍼티를 지닌다.
let printUser: PrintUserInfo<User> = (user) => console.log(user.id);

// Member는 id, nickname 프로퍼티를 지닌다.
let printMember: PrintUserInfo<Member> = (user) => {
  console.log(user.id, user.nickname); // nickname까지 출력
};

printMember = printUser; // ✅ OK.
printUser = printMember; // ❌ Error - Property 'nickname' is missing in type 'User' but required in type 'Member'.
```

printUser는 PrintUserInfo<User> 타입이기 때문에 Member 타입을 매개변수로 받을 수 없다. → printMember 함수를 printUser 변수에 할당할 수 X다.

? 근데 PrintUserInfo의 제네릭에 `extends User` 라고 되어있는데, Member 타입을 넘겨줘도 되는걸까?

O

Member가 User의 모든 속성을 포함하고 있기 때문이다!

```
interface User {
  id: string;
}

interface Member extends User {
  nickname: string;
}

// User = Member OK
// Member = User Error
```

✓ 객체의 메서드 타입을 정의하는 방법 2가지

🌟 `--strict모드` 에서 onChangeA같이 함수 타입을 화살표 표기법으로 작성하면, `반공변성` 을 띠게 된다.

🌟 onChangeB와 같이 함수 타입을 지정하면, 공변성과 반공변성을 모두 가지는 이변성을 띠게 된다.

→ 안전한 타입 가드를 위해 일반적으로 반공변적인 함수 타입을 설정하는 것이 권장된다.

```
interface Props<T extends string> {
  onChangeA?: (selected: T) => void; // 방법 1 : 반공변성
  onChangeB?(selected: T): void; // 방법 2 : 공변성, 반공변성(이변성)
}

const Component = () => {
  // 매개변수가 apple일 때 실행되는 메서드
  const changeToPineApple = (selectedApple: "apple") => {
    console.log("this is pine" + selectedApple);
  };

  return (
    <Select
      // 🚨 Error
      // onChangeA={changeToPineApple}

      // ✅ OK
      onChangeB={changeToPineApple}
    />
  );
};
```

! onChangeA는 `반공변성` 만 가지므로 더 구체적인 타입 `apple` 을 일반적인 타입 `string` 에 할당할 수 X다!

부모 = 자식 이 안된다

```
type T extends string // 일반적 - 부모(좁은 타입)
type "apple" // 구체적 - 자식(넓은 타입)
```



공변성과 반공변성의 최종 정리

일반적인 타입들은 좁은 타입 `부모` `User` 에서 넓은 타입 `자식` `Member` 으로 할당 가능하다. → `공변성`

제네릭 타입을 지닌 함수는 넓은 타입 `자식` 에서 좁은 타입 `부모` 으로 할당 가능하다. → `반공변성`

- strict 모드의 함수 타입
 - 화살표 함수 형식 - 반공변성 (권장)
 - 메서드 형식 - 공변성/반공변성



8.3 정리

TS를 잘 활용하면 리액트 프로젝트를 더 안정적으로 운영할 수 있다.

또한 다양한 훅을 활용하여 컴포넌트 내부 동작을 구현할 때도 타입을 명확하게 지정함으로써 실수를 미리 방지할 수 있게 해준다.

```
// 최종적인 Select 컴포넌트
import React, { useState } from "react";
import styled from "styled-components";

// 1. 테마 객체 정의
const theme = {
  fontSize: {
    default: "16px",
    small: "14px",
    large: "18px",
  },
  color: {
    white: "#FFFFFF",
    black: "#000000",
  },
};

// 2. 타입 할당
type Theme = typeof theme;
```

```

type FontSize = keyof Theme["fontSize"];
type Color = keyof Theme["color"];

// 3. 스타일 Props 인터페이스
interface SelectStyleProps {
  color: Color;
  fontSize: FontSize;
}

// 4. styled component
const StyledSelect = styled.select<SelectStyleProps>`
  color: ${({ color }) => theme.color[color]};
  font-size: ${({ fontSize }) => theme.fontSize[fontSize]};
`;

// 5. Select 컴포넌트 Props
type ReactSelectProps = React.ComponentPropsWithoutRef<"select">

interface SelectProps<OptionType extends Record<string, string>>
  extends Partial<SelectStyleProps> {
  id?: ReactSelectProps["id"];
  className?: ReactSelectProps["className"];
  options: OptionType;
  selectedOption?: keyof OptionType;
  onChange?: (selected?: keyof OptionType) => void;
}

// 6. Select 컴포넌트 구현
// 제네릭 타입 선언
// props 구조분해할당과 타입 지정
// 암시적으로 JSX.Element 반환
const Select = <OptionType extends Record<string, string>>({
  // 제네릭에 들어가면 자동으로 타입이 된다!
  // OptionType - { apple: string, banana: string, blueberry: string, ... }
  className,
  id,
  options,
  onChange,
  selectedOption,
  fontSize = "default",

```

```

    color = "black",
  }: SelectProps<OptionType>) => {
    const handleChange: React.ChangeEventHandler<HTMLSelectElement> =
      (e) => {
        const selected = Object.entries(options).find(
          ([_, value]) => value === e.target.value
        )?.[0];
        onChange?.(selected);
      };

    return (
      <StyledSelect
        id={id}
        className={className}
        fontSize={fontSize}
        color={color}
        onChange={handleChange}
        value={selectedOption && options[selectedOption]}
      >
        {Object.entries(options).map(([key, value]) => (
          <option key={key} value={value}>
            {value}
          </option>
        ))}
      </StyledSelect>
    );
  };
};

```

// 7. 실제 사용

```
const fruits = { apple: "사과", banana: "바나나", blueberry: "블루베리" }
```

```
type Fruit = keyof typeof fruits;
```

```
const FruitSelect = () => {
  const [fruit, changeFruit] = useState<Fruit | undefined>();

```

```

  return (
    <Select
      className="fruitSelectBox"
      options={fruits} // 여기서 OptionType이 typeof fruits로 결정됨
    />
  );
};

```

```
        onChange={changeFruit}
        selectedOption={fruit}
        fontSize="large"
      />
    );
  };

export default FruitSelect;
```

? OptionType은 뭐가 될까?

Select 컴포넌트의 options 속성에 fruits `{ apple: "사과", banana: "바나나", blueberry: 루베리" }`를 넣으면,
Select의 제네릭인

`{ apple: string, banana: string, blueberry: string }` OptionType은 가 된다.

🌟 즉, TS는 fruits 객체의 타입을 보고 자동으로 OptionType을 추론하는 것이다!

```
const fruits = { apple: "사과", banana: "바나나", blueberry:
// ...
return (
  <Select
    className="fruitSelectBox"
    options={fruits} // 여기서 OptionType이 typeof fruits이
    onChange={changeFruit}
    selectedOption={fruit}
    fontSize="large"
  />
```

```
const Select = <OptionType extends Record<string, string>>
// 제네릭에 들어가면 자동으로 타입이 된다!
// OptionType - { apple: string, banana: string, blueberry
  className,
  id,
  options,
  onChange,
  selectedOption,
  fontSize = "default",
  color = "black",
}: SelectProps<OptionType>) => {
```