5장 타입 활용하기

💠 조건부 타입

조건부 타입을 사용해 조건에 따라 출력 타입을 다르게 도출한다.

```
Condition ? A : B
```

- extends, infer, never

`extends` 와 제네릭을 활용한 조건부 타입

✔️ **extends 활용**

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

▼ `extends`의 제네릭의 한정자 역할

제네릭 타입 파라미터가 특정 타입이나 인터페이스를 만족해야 한다는 제약을 걸 때 용된다.

- 기본적인 한정자 사용

```
function getLength<T extends {length: number}>(arg: T):
    return arg.length; // T는 반드시 length 프로퍼티를 가져
}

getLength("string");       // string은 length 프로퍼티가 
getLength([1, 2, 3]);     // 배열도 length 프로퍼티가 있다
getLength({ length: 10 }); // length 프로퍼티가 있는 객체

getLength(123); // 🚨 number 타입은 length 프로퍼티가 없다
```

- 여러 타입을 조합한 한정자

```
interface HasName {
    name: string;
}

interface HasAge {
    age: number;
}

// T는 반드시 HasName과 HasAge를 모두 만족해야 한다
function printNameAndAge<T extends HasName & HasAge>(ob
    console.log(`${obj.name} is ${obj.age} years old`);
}
```

- keyof와 함께 사용

```
function getProperty<T, K extends keyof T>(obj: T, k
    return obj[key];
}

const person = { name: "John", age: 30 };
const name = getProperty(person, "name");  // "John"
```

```
const age = getProperty(person, "age");    // 30

const invalid = getProperty(person, "invalid");  //
```

```
T extends U ? X : Y // 삼항 연산자 사용
// T를 U에 할당할 수 있으면 X타입, 아니면 Y타입
```

```
interface Bank {
    financialCode: string;
    companyName: string;
    name: string;
    fullName: string;
}

interface Card {
    financialCode: string;
    companyName: string;
    name: string;
    appCardType?: string;
}

// extends 키워드는 일반적으로 문자열 리터럴과 함께 사용하지 않는다.
type PayMethod<T> = T extends "card" ? Card : Bank;
type CardPayMethodType = PayMethod<"card">;
type BankPayMethodType = PayMethod<"bank">;
```

➢ 조건부 타입을 사용하지 않았을 때의 문제점

✔ react-query 활용 예시

- 계좌, 카드, 앱 카드 3가지 결제 수단 정보를 가져오는 API가 있다.
- API의 엔드포인트는 각각 다르다.

```
계좌 정보 엔드포인트: www.baemin.com/baeminpay/.../bank
카드 정보 엔드포인트: www.baemin.com/baeminpay/.../card
앱 카드 정보 엔드포인트: www.baemin.com/baeminpay/.../appcard
```

- 각 API는 정보를 배열 형태로 반환한다.

> ⛳ 3가지 API의 엔드포인트가 비슷하기 때문에 서버 응답을 처리하는 공통 함수를 생성하고,
> 해당 함수에 타입을 전달하여 타입별로 처리 로직을 구현하자.

타입

- 사용되는 타입 정리
  - PayMethodBaseFromRes: 서버에서 받아오는 결제 수단 기본 타입으로 은행과 카드에 모두 들어가 있다.
  - Bank, Card: 은행과 카드 각각에 맞는 결제 수단 타입이다. 결제 수단 기본 타입인 PayMethodBaseFromRes를 상속받아 구현한다.
  - PayMethodInterface: 프론트에서 관리하는 결제 수단 관련 데이터로 UI를 구현하는 데 사용되는 타입이다.
  - PayMethodInfo<T extends Bank | Card>:
    - 최종적인 은행, 카드 결제 수단 타입이다. 프론트에서 추가되는 UI 데이터 타입과 제네릭으로 받아오는 Bank 또는 Card를 합성한다.
    - extends를 제네릭에서 한정자로 사용하여 Bank 또는 Card를 포함하지 않는 타입은 제네릭으로 넘겨주지 못하게 방어한다.
    - BankPayMethodInfo = PayMethodInterface & Bank처럼 카드와 은행의 타입을 만들어줄 수 있지만 제네릭을 활용해서 중복된 코드를 제거한다.

```
// 서버에서 받아오는 결제 수단 기본 타입
interface PayMethodBaseFromRes {
    financilCode: string;
    name: string;
}

interface Bank extends PayMethodBaseFromRes {
    fullName: string;
}

interface Card extends PayMethodBaseFromRes {
```

```
    appCardtype?: string;
}

// 프론트에서 관리하는 결제 수단 관련 데이터로 UI를 구현하는 데 사용하는 타입
type PayMethodInterface = {
    companyName: string;      // 결제 회사명 (예: "비자", "마스터카드"
    cardNumber?: string;      // 카드 번호
    cardType?: string;        // 카드 종류 (예: "신용카드", "체크카드"
    // ..
}

// 최종적인 은행, 카드 결제 수단 타입
type PayMethodInfo<T extends Bank | Card> = T & PayMethodInterfa
```

react-query의 useQuery를 사용하여 구현한 커스텀 훅 : `useGetRegisteredList` 함수

- 반환값 : useQuery의 반환값
- `useCommonQuery<T>` : useQuery를 한 번 래핑해서 사용하고 있는 함수. useQuery의 반환 data를 T 타입으로 반환한다.
- `fetcherFactory` : axios를 래핑해주는 함수. 서버에서 데이터를 받아온 후 onSuccess 콜백함수를 거친 결괏값을 반환한다.

```
type PayMethodType = PayMethodInfo<Card> | PayMethodInfo<Bank>;

/*
* 타입으로 "card", "appcard", "bank" 받아서 해당 결제 수단의 결제 수단 정
*/
export const useGetRegisteredList = (
    type: "card" | "appcard" | "bank"
): UseQueryResult<PayMethodType[]> => {
    const url = `baeminpay/codes/${type == "appcard" ? "card" :

    const fetcher = fetcherFactory<PayMethodType[]>({
        onSuccess: (res) => {
            const usablePocketList =
                res?.filter(  // useType이 "USE"인 항목만 선택
                    (pocket: PocketInfo<Card> | PocketInfo<Bank>
                        pocket?.useType === "USE"
                ) ?? [];
            return usablePocketList;
```

```
        },
    });


    const result = useCommonQuery<PayMethodType[]>(url, undefine
    // undefined를 명시적으로 전달하는 것은 "이 API 호출에는 파라미터가 필

    return result;
};
```

✏️ `fetcherFactory` 부분 코드를 분석해보자.

fetcherFactory 함수는 `onSuccess 프로퍼티를 가진` 객체 리터럴을 매개변수로 받고있다.

onSuccess는 콜백함수이다.

- 콜백함수인 이유
  1. fetcherFactory 함수에 **인자**로 전달되고 있고
  2. fetcherFactory 함수의 실행이 끝난 뒤(API 호출이 성공한 뒤)에 실행되기 때문이다.

```
const fetcher = fetcherFactory<PayMethodType[]>({
    onSuccess: (res) => {
        const usablePocketList =
            res?.filter(
                (pocket: PocketInfo<Card> | PocketInfo
                    pocket?.useType === "USE"
            ) ?? [];
        return usablePocketList;
    },
});
```

❓ ▼ 그렇다면 fetcherFactory 함수는 어떻게 설계되어 있는걸까? (예시)

- 고차함수 사용

1. fetcherFactory 호출
   - 여기서 `onSuccess` 콜백 함수를 포함한 옵션 객체를 전달
   - fetcherFactory는 새로운 함수를 반환하여 fetcher에 저장 `옵션을 기억` `함수를 생성`

```
const fetcher = fetcherFactory<PayMethodType[]>({
    onSuccess: (res) => {
        // 필터링 로직
        return usablePocketList;
    }
});
```

2. useCommonQuery에서 fetcher 사용
   - useCommonQuery 내부에서 fetcher 함수가 실행
   - 이때 fetcher는 API 호출을 수행하는 함수로 동작하고, API 호출 성공
     onSuccess가 실행된다.

```
interface FetcherOptions<T> {
  onSuccess?: (response: T) => T; // 성공 시 실행될 콜백함수
  onError?: (error: any) => void; // 에러 시 실행될 콜백함수
  // 기타 필요한 옵션들...
}

// T는 API 응답 데이터의 타입
function fetcherFactory<T>(options: FetcherOptions<T>)
    // API 호출하는 함수를 반환
    return async (url: string, params?: any) => {
        try {
            // API 호출
            const response = await axios.get(url, { par

            // 성공 콜백이 있으면 실행하고 그 결과를 반환
            if (options.onSuccess) {
                return options.onSuccess(response.data)
            }
```

```
            // 없으면 그냥 데이터 반환
            return response.data;

        } catch (error) {
            // 에러 콜백이 있으면 실행
            if (options.onError) {
                options.onError(error);
            }
            throw error;
        }
    };
}
```

▼ 고차함수 : 함수를 반환하는 함수

```
function multiply(x: number) {
    // x값을 기억하는 새로운 함수를 반환
    return function(y: number) {
        return x * y;
    }
}

// multiply(2)를 호출하면:
// - x가 2인 새로운 함수가 반환됨
// - 이 함수를 multiplyByTwo에 저장
const multiplyByTwo = multiply(2);

// multiplyByTwo(4)를 호출하면:
// - 저장된 함수가 y=4로 실행됨
// - x=2와 y=4를 곱해서 8을 반환
console.log(multiplyByTwo(4));  // 8
```

~~(p.153) 이때 useGetRegisteredList 함수가 반환하는 Data 타입은 PocketInfocCard〉 : PocketInfoBank>이다. 사용자가 타입으로 "card"를 넣었을 때 useGetRegisteredList 함수가 반환하는 Data 타입은 PocketInfo 라고 유추할 수 있다. 하지만 useGetRegisteredList 함~~

~~수가 반환하는 Data 타입은 PayMethodType 이기 때문에 사용하는 쪽에서는 PocketInfo일 가능성도 있다.~~

```
const { data: pocketList } = useGetRegisteredList("card");
// PayMethodInfo<Card> | PayMethodInfo<Bank> 타입을 가진 데이터
```

이때 useGetRegisteredList 함수가 반환하는 타입이 PayMethodType이다.

❗ 하지만 반환 타입이 유니온이라서 "card"를 입력해도 Typescript에서는 반환값이 카드일 수도 있고 은행일 수도 있다고 생각하여 에러가 발생한다.

🙍‍♀️ 그래서 제네릭과 조건부 타입을 사용해서 타입 시스템에게 `"card를 입력하면 무조건 Card 타입이 반환된다"` 는 것을 알려줘야 한다.

➢ extends 조건부 타입을 활용하여 개선하기

useGetRegisteredList 함수의 반환 Data는 인자 타입에 따라 정해져 있다.

```
type: "card" | "appcard" -> PocketInfo<Card>
type: "bank" -> PocketInfo<Bank>
```

> 계좌와 카드의 API 함수를 각각 만들 수도 있지만, 엔드포인트의 마지막 경로만 다르고 계와와 카드가 같은 컴포넌트에서 사용되기 때문에 하나의 함수에서 한번에 관리해야 하는 상황이다.
>
> ⛳
> **조건부 타입**을 활용하여 하나의 API 함수에서 타입에 따른 정확한 반환 타입을 추론하게 해보자!

```
// type PayMethodType = PayMethodInfo<Card> | PayMethodInfo<Bank

type PayMethodType<T extends "card" | "appcard" | "bank"> =
  T extends "card" | "appcard"
    ? PayMethodInfo<Card>
    : PayMethodInfo<Bank>;

// T가 "card"나 "appcard"면 → PayMethodInfo<Card> 타입을
// T가 "bank"면 → PayMethodInfo<Bank> 타입을 반환
```

```
// export const useGetRegisteredList = (
//  type: "card" | "appcard" | "bank"
// ): UseQueryResult<PayMethodType[]> => {


export const useGetRegisteredList = <T extends "card" | "appcard
    type: T
): UseQueryResult<PayMethodType<T>[]> => {
    const url = `baeminpay/codes/${type === "appcard" ? "card" :

    // ...

    const result = useCommonQuery<PayMethodType<T>[]>(url, undef

    return result;
};
```

`PayMethodType`이 사용자가 인자에 넣는 타입 값에 맞는 타입만을 반환하도록 구현했다. 이제 인자로 “card” 또는 “appcard”를 넣으면 PocketInfo<Card>를 반환하고, “bank”를 넣으면 PocketInfo<Bank>를 반환한다.

이로써 사용자는 useGetRegisterList 함수를 사용할 때 불필요한 타입 가드를 하지 않아도 된다.

우리가 다룬 extends 활용 사례 정리

1. 제네릭과 extends를 함께 사용해 제네릭으로 받는 타입을 제안했다. → 개발자는 잘못된 값을 넘길 수 없기 때문에 휴먼 에러를 방지한다.
2. extends를 활용해 조건부 타입을 설정했다. 조건부 타입을 사용해서 반환값을 사용자가 원하는 값으로 구체화할 수 있다.
   → 불필요한 타입 가드, 타입 단언 등을 방지한다.

`infer`를 활용해서 타입 추론하기

`infer` : 타입을 추론하는 역할

extends를 사용할 때 `infer` 키워드를 사용할 수 있다.

- 삼항 연산자를 사용한 조건문의 형태를 가지는데, `extends`로 조건을 서술하고 `infer`로 타입을 추론하는 방식이다.

```
type UnpackPromise<T> = T extends Promise<infer K>[] ? K : any;

// T가 Promise로 래핑된 경우라면 K를 반환하고, 아닐 경우 any를 반환한다.
// Promise<infer K> - Promise의 반환값을 추론해 해당 값의 타입을 K로 한

const promises = [Promise.resolve("Mark"), Promise.resolve(38)];
type Expected = UnpackPromise<typeof promises>; // string | numb

// typeof promieses: Promise<string> | Promise<number>[] 타입
// Promise<infer K>[]: Promise 배열인지 확인하고, 각 Promise의 결과 E
// infer K: Promise 안의 타입을 추론해서 K에 저장
```

- extends, infer, 제네릭을 활용하면 타입을 조건에 따라 더 세밀하게 사용할 수 있다.

✔️ 예시) 배민 라이더를 관리하는 라이더 어드민 서비스에서 사용하는 타입

- RouteBase, RouteItem : 라이더 어드민에서 라우팅을 위해 사용하는 타입. 배열 형태로 사용되며, 권한 API로 반환된 사용자 권한과 `name`을 비교하여 인가되지 X은 접근을 방지한다.
    - RouteItem의 `name`은 pages가 있을 때는 단순히 이름의 역할만 하고, 그렇지 않을 때는 사용자 권한과 비교한다.

> 라우팅 : 웹 애플리케이션에서 사용자가 URL을 통해 다른 페이지로 이동하거나, 다른 경로에 대한 요청을 처리하는 방법 정의

```
// 중첩된 라우트 구조를 위한 interface : RouteBase, RouteItem

interface RouteBase { // 기본 라우트 정보
    name: string;
    path: string;
    component: ComponentType;
}

export interface RouteItem { // 기본 라우트 정보 + 하위 페이지
```

```
    name: string;
    path: string;
    component?: ComponentType;
    pages?: RouteBase[];
}

/* 실제 페이지 라우팅을 담당 */
export const routes: RouteItem[] = [
    {
        name: "기기 내역 관리", // 사용자 권한
        path: "/device-history",
        component: DeviceHistoryPage,
    },
    {
        name: "헬멧 인증 관리",
        path: "/helmet-certification",
        component: HelmetCertificationPage,
    },

    // ...

    // 중첩된 라우트 구조 예시
   {
      name: "계정 관리", // 단순히 이름의 역할만
      path: "/account",
      pages: [
          {
              name: "기기 내역 관리", // 사용자 권한
              path: "/device-history",
              component: DeviceHistoryPage
          },
          {
              name: "헬멧 인증 관리",
              path: "/helmet-certification",
              component: HelmetCertificationPage
          }
      ]
  },
]
```

- MainMenu, SubMenu : 메뉴 리스트에서 사용하는 타입. 권한 API를 통해 반환된 사용자 권한과 name을 비교하여, 사용자가 접근할 수 있는 메뉴만 렌더링한다.
    - MainMenu의 `name`은 subMenus를 가지고 있을 때 단순히 이름의 역할만 하며, 그렇지 않을 때는 권한으로 간주된다.

```
export interface SubMenu {
    name: string;
    path: string;
}

export interface MainMenu {
    name: string;
    path?: string;
    subMenus?: SubMenu[];
}

export type MenuItem = MainMenu | SubMenu;

/* 사용자 인터페이스에 보여질 네비게이션 메뉴 구조 */
export const menuList: MenuItem[] = [
    {
        name: "계정 관리", // 그룹용 메뉴 : 단순히 이름의 역할 (권한 검사
        subMenus: [
            {
                name: "기기 내역 관리", // 실제 페이지로 이동할 메뉴
                path: "/device-history",
            },
            {
                name: "헬멧 인증 관리",
                path: "/helmet-certification",
            },
        ]
    },
    {
        name: "운행 관리", // 권한
        path: "/operation",
    },
    // ...
];
```

❗ `routes`와 `menuList`에서 사용되는 `name`이 서로 일치해야 하는데, 둘 다 단순히 `string` 타입으로 되어있어서 실수로 다른 값을 입력해도 타입 체크에서 걸러지지 않는다.

🙍‍♀️ 별도 타입 `PermissionNames` 을 선언하여 name을 관리하는 방법이 있다.

```
type PermissionNames = "기기 정보 관리" | "안전모 인증 관리" | "운영 여
```

하지만 권한 검사가 필요 없는 subMenus나 pages가 존재하는 name은 따로 처리해야 한다.
→ `infer`, 불변객체`as const` 를 활용하여 menuList 또는 routes의 값을 추출하여 타입으로 정의한다.

💡 코드 작성 흐름

1. menuList를 `as const`로 정의한다.
2. menuList에서 name들을 추출해서 `PermissionNames` 타입을 생성한다. `infer`

   나는 PermissionNames에 실제 페이지로 이동하는 메뉴들의 name을 담았다.

   즉, menuList에 subMenus가 있을 경우에는 subMenus의 name이 필요하고, subMenus가 없을 경우에는 MainMenu의 name이 필요하다.

   ▼ menuList 구조

```
const menuList = [
    {
        name: "계정 관리",        // subMenus가 있으므로
        subMenus: [
            {
                name: "기기 내역 관리",    // 이건 포함
                path: "/device-history",
            },
            {
                name: "헬멧 인증 관리",    // 이건 포함
                path: "/helmet-certification",
            },
        ]
    },
    {
        name: "운행 관리",        // subMenus가 없으므로
        path: "/operation",
    }
] as const;
```

```
type ExtractNames<T> = T extends readonly (
    | { name: infer N; subMenus?: undefined }  // subMe
    | { name: string; subMenus: readonly { name: infer
                    // subMenus 배열 안의 name 값들을 N으로
)[]
    ? N
    : never;
```

```
type PermissionNames = ExtractNames<typeof menuList>;
// type PermissionNames = "기기 내역 관리" | "헬멧 인증 관리
```

3. 이 PermissionNames를 사용해서 RouteItem 타입을 정의한다.

```
// <menuList 값을 추출하는 예시>

export interface MainMenu {
    // ...

    // subMenus?: SubMenu[];
    subMenus?: ReadonlyArray<SubMenu>;
}

export const menuList = [
    // ...
] as const; // 불변 객체로 만들어서 타입을 추출한다.


// 라우트 부분 수정하기!
interface RouteBase {
    // name: string;
    name: PermissionNames;
    path: string;
    component: ComponentType;
}

export type RouteItem = { /* 그룹용 메뉴 ex) 계정관리 */
    name: string; // 그룹용 라우트 (권한 검사가 필요없는 경우)
    path: string;
    component?: ComponentType;
    pages: RouteBase[]; // 그룹 안에 있는 메뉴가 라우트의 name과 일치할
    } | { /* 실제 페이지용 메뉴 ex) 기기내역관리 */
        name: PermissionNames; // (실제 페이지)권한 검사가 필요한 경우
        path: string;
        component?: ComponentType;
    };
```

➡️ 즉, 이렇게 하면 실제 페이지로 이동하는 메뉴와 반드시 라우트의 name이 일치해야 하고, 단순히 하위 메뉴를 그룹화하는 용도의 라우트는 아무 이름이나 사용할 수 있게 된다.

✔️ 내가 예시로 구현했던 ExtractNames의 타입의 다른 구현 방식이다. (책에 나와있는지 모르고 내가 위에서 PermissionNames 타입을 만드는 예시를 만들었다.)

- MenuItem 배열을 받아서
- MainMenu인 경우
  - subMenus가 있으면 → 재귀적으로 subMenus의 name들을 추출
  - subMenus가 없으면 → MainMenu의 name을 추출
- SubMenu인 경우 → SubMenu의 name을 추출

```
type UnpackMenuNames<T extends ReadonlyArray<MenuItem>> =
  // 1. T는 MenuItem 배열이어야 함
  T extends ReadonlyArray<infer U>
    // 2. U가 MainMenu 타입이면
    ? U extends MainMenu
      // 3. U의 subMenus를 V로 추출
      ? U["subMenus"] extends infer V
        // 4. V가 SubMenu 배열이면
        ? V extends ReadonlyArray<SubMenu>
          // 5. 재귀적으로 V(subMenus)의 names를 추출
          ? UnpackMenuNames<V>
          // 6. SubMenu 배열이 아니면 MainMenu의 name
          : U["name"]
        : never
      // 7. U가 SubMenu 타입이면
      : U extends SubMenu
        // 8. SubMenu의 name
        ? U["name"]
        : never
    : never;
```

💠 템플릿 리터럴 타입 활용하기

유니온 타입을 사용하여 변수 타입을 특정 문자열로 지정할 수 있다.

```
type Headertag = "h1" | "h2" | "h3" | "h4" | "h5";
```

- 컴파일 타임의 변수에 할당되는 타입을 특정 문자열로 정확하게 검사하여 휴먼 에러 방지
- 자동 완성 기능을 통한 개발 생산성 up

✏️ TS 4.1부터 이를 확장하는 방법인 템플릿 리터럴 타입을 지원하기 시작했다.

`템플릿 리터럴 타입` : JS의 템플릿 리터럴 문법을 사용해 특정 문자열에 대한 타입을 선언할 수 있는 기능

```
type HeadingNumber = 1 | 2 | 3 | 4 | 5;
type HeaderTag = `h${HeadingNumber}`;
```

- 더욱 읽기 쉬운 코드로 작성 가능
- 재사용, 수정하는 데 용이한 타입 선언 가능

✔️ 배민외식업광장 예시

Direction 타입은 “top”, “bottom”, “left”, “right”가 합쳐진 문자열로 선언되어 있다.

```
type Direction =
    | "top"
    | "topLeft"
    | "topRight"
    | "bottom"
    | "bottomLeft"
    | "bottomRight";
```

이 코드에 템플릿 리터럴 타입을 적용하면 좀 더 명확하게 표시 가능하다.

```
type Vertical = "top" | "bottom";
type Horizon = "left" | "right";

type Direction = Vertical | `${Vertical}${Capitalize<Horizon>}`;
```

💠 커스텀 유틸리티 타입 활용하기

표현하기 힘든 타입을 마주할 경우 TS에서 제공하는 유틸리티 타입만으로는 표현하는 데 한계를 느낀다.

→ 유틸리티 타입을 활용한 `커스텀 유틸리티 타입`을 제작해서 사용하자

유틸리티 함수를 활용해 styped-components의 중복 타입 선언 피하기

ex) 컴포넌트의 background-color, size 값을 props로 받아와서 상황에 맞게 스타일을 구현할 때가 많다.

스타일 관련 props는 styled-component에 전달되며, styled-components에도 해당 타입을 정확하게 작성해줘야 한다.

→ `Pick`, `Omit` 같은 유틸리티 타입을 잘 활용하여 코드를 간결하게 작성해보자

Props 타입과 styled-components 타입의 중복 선언 및 문제점

이때 Pick을 사용하지 않으면, StyledProps를 정의할 때 Props와 똑같은 타입임에도 똑같이 다시 작성해야 한다. → `Pick`을 활용해 props에서 필요한 부분만 선택하여 styled-components 컴포넌트의 타입을 정의하자

```
// HrComponent.tsx
export type Props = {
    height?: string;
    color?: keyof typeof colors;
    isFull?: boolean; // 화면 좌우 기본 패딩값 무시하고 수평선을 꽉 차게
    className?: string;
    ...
}

export const Hr: FC<Props> = ({ height, color, isFull, classNam
    ...
    return <HrComponent height={height} color={color} isFull={is
}


// style.ts
import { Props } from './HrComponent';

type StyledProps = Pick<Props, "height" | "color" | "isFull">;
```

```
const HrComponent = styled.hr<StyledProps>`
    height: ${({ height }) => height || "10px"};
    marign: 0;
    background-color: ${({ color }) => colors[color || "gray7"]};
    border: none;
    ${({ isFull }) =>
        isFull &&
            css`
                margin: 0 -15px;
            `
    }
`
```

▼ `Pick`

: 기존 타입에서 특정 속성만 선택해서 새로운 타입을 만드는 유틸리티 타입

```
Pick<타입, "선택할속성1" | "선택할속성2" | ...>
```

```
// 원본 타입
interface Person {
    name: string;
    age: number;
    address: string;
    phone: string;
}

// Pick 사용
type PersonNameAge = Pick<Person, "name" | "age">;

// PersonNameAge는 이렇게 됨:
// {
//     name: string;
//     age: number;
// }

// 하나만 선택
type JustName = Pick<Person, "name">;

// 여러 개 선택
type ContactInfo = Pick<Person, "phone" | "address">;

// 변수에 타입 지정
const person: Pick<Person, "name" | "age"> = {
    name: "Kim",
    age: 25
    // address, phone은 포함할 수 없음
};
```

❓ TS에서 styled-component를 사용할 때 무조건 타입을 지정해줘야 할까?
X

props를 사용하는 경우에 타입 지정이 필요하다.

```
// props를 사용하는 경우
const Button = styled.button<{ primary?: boolean }>`
  color: ${props => props.primary ? 'red' : 'black'};
`;
```

PickOne 유틸리티 함수

서로 다른 2개 이상의 객체를 유니온 타입으로 받을 때 타입 검사가 제대로 진행되지 않는 이슈가 있다.

→ `PickOne` 유틸리티 함수

```
type Card = {
    card: string
};
type Account = {
    account: string
};

function withdraw(type: Card | Account) {
    ...
}

withdraw({ card: "hyundai", account: "hana" }); // 타입 에러가 발생
```

❗ card, account 속성을 모두 받아도 타입 에러가 발생하지 않는다. 집합 관점으로 볼 때 유니온은 합집합이 되기 때문이다.

식별할 수 있는 유니온으로 객체 타입을 유니온으로 받기

각 타입에 type이라는 공통된 속성을 추가하여 구분한다.

```
type Card = {
    type: "card";
    card: string;
};
type Account = {
    type: "account";
    account: string;
};

function withdraw(type: Card | Account) {
    ...
}

withdraw({ type: "card", card: "hyundai" });
withdraw({ type: "account", card: "hana" });
```

❗ 일일이 type을 다 넣어줘야 하는 불편함

PickOne 커스텀 유틸리티 타입 구현하기

> account일 때는 card를 받지 못하고, card일 때는 account를 받지 못하게 해야한다.
> → 하나의 속성이 들어왔을 때 다른 타입을 옵셔널한 undefined 값으로 지정하는 방법은 어떨까?

1. `옵셔널 + undefined`

`옵셔널 + undefined`로 타입을 지정하면, 사용자가 의도적으로 undefined 값을 넣지 않는 이상, 원치 않는 속성에 값을 넣었을 때 타입 에러가 발생한다.

```
{account: string; card?: undefined} | {account?: undefined; carc
```

- account, card, payMoney 속성 중 하나만을 필수로 받는 payMoney 속성을 추가해보자

```
type PayMethod =
  | { account: string; card?: undefined; payMoney?: undefinec
```

```
    | { account?: undefined; card: string; payMoney?: undefinec
    | { account?: undefined; card?: undefined; payMoney: number
```

➡️ 즉, 선택하고자 하는 하나의 속성을 제외하고, 나머지 값을 `옵셔널타입`?` + undefined`로 설정하면 원하고자 하는 속성만 받을 수 있다.

2. `커스텀 유틸리티 타입`

이를 `커스텀 유틸리티 타입`으로 구현해보자!

```
type PickOne<T> = {
    [P in keyof T]: Record<P, T[P]> & Partial<Record<Exclude<key
}[keyof T];
```

PickOne 살펴보기

```
// 위 커스텀 유틸리티 타입 구현 코드
type PickOne<T> = {
    [P in keyof T]: Recourd<P, T[P]> & Partial<Record<Exclude<ke
}[keyof T];
```

T에는 객체가 들어 온다고 가정한다.

- One<T>

```
// 예시1
interface Example {
   a: string;
   b: number;
}
///////

// 예시2
type Card = {card: string};

type One<T> = { [P in keyof T]: Record<P, T[P]> }[keyof T];

{ [P in keyof Card]: Record<P, Card[P]> }[keyof Card]
```

1. `[ P in keyof T ]`에서 T는 객체로 가정하기 때문에 `P`는 `T 객체의 키값`을 말한다.

```
// 예시1
'a'
'b'
// 예시2
'card'
```

2. `Record<P, T[P]>`는 P 타입을 키로 가지고, value는 P를 키로 둔 T 객체의 값의 레코드 타입을 말한다.
3. 따라서 `{ [P in keyof T]: Record<P, T[P]> }` 에서 키는 T 객체의 키 모음이고, value는 해당 키의 원본 객체 T를 말한다.

```
// 예시1
{
    a: Record<'a', string>;  // { a: string }
    b: Record<'b', number>;  // { b: number }
}
// 예시2
{
    card: Record<'card', string>  // { card: string }
}
```

4. 3번의 타입에서 다시 [keyof T]의 키값으로 접근하면 다음과 같다.

   (최종 결과는 전달받은 T와 같다.)

```
// 예시1
// keyof T는 'a' | 'b'이므로
// Record<'a', string> | Record<'b', number>
// 이건 결국 { a: string } | { b: number } 이다.

// 예시2
// => { card: Record<'card', string> }['card']
// => Record<'card', string>
// => { card: string }
```

- ExcludeOne<T>

```
// type One<T> = { [P in keyof T]: Record<P, T[P]> }[keyof T]
type ExcludeOne<T> = { [P in keyof T]: Partial<Record<Exclude
```

1. `[ P in keyof T ]`에서 T는 객체로 가정하기 때문에 `P`는 `T 객체의 키값`을 말한다.
2. `Exclude<keyof T, P>` 는 T 객체가 가진 키값에서 P 타입과 일치하는 키값을 제외한다.

   이 타입을 A라 하자.
3. `Record<A, undefiend>`는 키로 A 타입을, 값으로 undefined 타입을 갖는 레코드 타입이다. → 전달받은 객체 타입을 모두 `{ [key]: undefined }`형태로 만든다.

   이 타입을 B라 하자.
4. `Partial<B>` 는 B 타입을 옵셔널로 만든다. → `{ [key]?: undefined }`
5. 최종적으로 [P in keyof T]로 매핑된 타입에서 동일한 객체의 키값인 [keyof T]로 접근하기 때문에 4번 타입이 반환된다.

▼ `Exclude`

```
Exclude<UnionType, ExcludeType> // 첫번째 유니온 타입에서
```

```
// 예시
type T1 = Exclude<"a" | "b" | "c", "a">;  // "b" | "
type T2 = Exclude<string | number | boolean, string>

// 실제 사용 예
type Animals = "dog" | "cat" | "bird" | "fish";
type Pets = Exclude<Animals, "fish">;  // "dog" | "c
```

▼ `Partial`

타입의 모든 속성을 선택적으로 만든다.

```
// 예시
interface User {
    name: string;
    age: number;
}

type PartialUser = Partial<User>;
// 결과:
// {
//    name?: string;
//    age?: number;
// }
```

➡️ 결론적으로 얻고자 하는 타입이 `속성 하나와 나머지는 옵셔널+undefined인 타입`이기 때문에 앞의 속성을 활용해서 PickOne 타입을 표현할 수 있다.

- PickOne<T>

```
type PickOne<T> = One<T> & ExcludeOne<T>;
// type One<T> = { [P in keyof T]: Record<P, T[P]> }[keyof T]
// type ExcludeOne<T> = { [P in keyof T]: Partial<Record<Excl
```

1. One<T> & ExcludeOne<T>는 `[P in keyof T]`를 공통으로 갖기 때문에 교차된다.

   ```
   [P in keyof T]: Record<P, T[P]> & Partial<Record<Exclude<k
   ```

   타입 해석 : 전달된 T 타입의 1개의 키는 값을 가지고 있으며, 나머지 키는 옵셔널한 undefined 값을 가진 객체를 의미한다.

```
type Card = {card: string};
type Account = {account: string};

type PickOne<T> = {
    [P in keyof T]:
```

```
        Record<P, T[P]> & // 현재 속성은 필수
        Partial<Record<Exclude<keyof T, P>, undefined>>; // L
}[keyof T];

const pickOne1: PickOne<Card & Account> = {card: "hyundai"};
const pickOne2: PickOne<Card & Account> = { account: "hana" }
const pickOne3: PickOne<Card & Account> = { card: "hyundai",
const pickOne4: PickOne<Card & Account> = { card: undefined,
const pickOne5: PickOne<Card & Account> = { card: "hyundai",
```

PickOne 타입 적용하기

```
type Card = {
    type: "card";
    card: string;
};
type Account = {
    type: "account";
    account: string;
};

type CardOrAccount = PickOne<Card & Account>;

// function withdraw(type: Card | Account) {
function withdraw(type: CardOrAccount) {
    ...
}

withdraw({ card: "hyundai", account: "hana" }); // 🚨 에러 발생
```

➡️ 한번에 커스텀 유틸리티 타입 함수를 작성하기 쉽지 않기 때문에 `커스텀 유틸리티 타입` 구현 시, 정확히 어떤 타입을 구현해야 하는지 파악하고, 필요한 타입을 작인 단위로 쪼개어 생각하여 단계적으로 구현하는 게 좋다.

NonNullable 타입 검사 함수를 사용하여 간편하게 타입 가드하기

`null을 가질 수 있는 값의 null 처리`는 자주 사용되는 타입 가드 패턴의 하나이다.

- 일반적으로 `if문`을 통해 null 처리 타입 가드를 적용하지만, `is` 키워드와 `NonNullable` 타입으로 타입 검사를 위한 유틸 함수를 만들어서 사용할 수도 있다.

NonNullable 타입이란

제네릭으로 받는 T가 `null` 또는 `undefined`일 때 `never` 또는 `T`를 반환하는 타입

- null이나 undefined를 제외한 타입을 만들어준다.

```
// string | null | undefined -> string
type T1 = NonNullable<string | null | undefined>;  // string

// number | undefined -> number
type T2 = NonNullable<number | undefined>;  // number

// null | undefined -> never
type T3 = NonNullable<null | undefined>;  // never
```

null, undefined를 검사해주는 NonNullable 함수

NonNullable 유틸리티 타입을 사용하여 null 또는 undeifined를 검사해주는 타입 가드 함수

```
function NonNullable<T>(value: T): value is NonNullable<T> {
    return value !== null && value !== undefined;
    // value가 null도 아니고 undefined도 아니면 true
    // value가 null이거나 undefined면 false
}
```

```
// 예시
let value: string | null | undefined = "hello";

if (NonNullable(value)) {
    // value가 null도 아니고 undefined도 아닐 때
    // 즉, string 타입이 된다.
    value  // ✅ string 타입
} else {
    // value가 null이거나 undefined일 때
    value  // never 타입
}
```

Promise.all을 사용할 때 NonNullable 적용하기

✔️ 예시) 각 상품 광고를 노출하는 API 함수 레이어

상품광고 API

- 상품광고 API는 상품 번호인 shopNumber 매개변수에 따라 각기 다른 응답 값을 반환하는 광고 조회 API이다.
- 여러 상품의 광고를 조회할 때 하나의 API에서 에러가 발생한다고 해서 전체 광고가 보이지 않으면 안된다.

  → try-catch문을 사용하여 에러가 발생할 때 null을 반환하고 있다.

```
class AdCampaignAPI {
    static async operating(shopNo: number): Promise<AdCampaign[]
        try {
            return await fetch(`/ad/shopNumber=${shopNo}`);
        } catch (error) {
            return null;
        }
    }
}
```

AdCampaignAPI를 사용해서 여러 상품의 광고를 받아오는 로직

Promise.all을 사용해 각 shop의 광고를 받아오고 있다.

- AdCampaignAPI.operating 함수에서 null을 반환할 수 있기 때문에 shopAdCampaignList 타입은 Array<AdCampaign[] | null>로 추론된다.

```
const shopList = [
    {shopNo: 100, category: "chicken"},
    {shopNo: 101, category: "pizza"},
    {shopNo: 102, category: "noodle"},
];

const shopAdCampaignList = await Promist.all(shopList.map((shop)
    => AdCampaignAPI.operating(shop.shopNo))
); // Array<AdCampaign[] | null>
```

📚 `Promise.all`

여러 개의 Promise를 병렬로 실행하고, 모든 Promise가 완료될 때까지 기다린 후 결과를 배열로 반환한다. ex) [Promise, Promise, Promise ..]

❗ shopAdCampaignList 변수를 `NonNullable` 함수로 필터링하지 않으면, 순회할 때 `map` 마다 고차 함수 내 콜백 함수에서 if문을 사용한 타입 가드를 반복하게 된다.

```
// 1. NonNullable 사용하지 않을 때
shopAdCampaignList.map(campaign => {
    if (campaign) {  // 매번 null 체크 필요
        console.log(campaign.id);
    }
});
```

👩 NonNullable로 shopAdCampaignList를 필터링하면, shopAds는 원하는 타입인 Array<AdCamaign[]>로 추론할 수 있다.

- (장점) NonNullable로 한 번 필터링하면:
  - null인 항목들이 모두 제거됨
  - 이후 작업에서는 타입 가드 불필요
  - 코드가 더 깔끔해짐

```
// 2. NonNullable로 필터링하면
const filteredList = shopAdCampaignList.filter(NonNullable);  //
filteredList.map(campaign => {
    console.log(campaign.id);  // null 체크 불필요
});
```

```
// 전체 코드

function NonNullable<T>(value: T): value is NonNullable<T> { //
    return value !== null && value !== undefined;
    // value가 null도 아니고 undefined도 아니면 true
    // value가 null이거나 undefined면 false
}

const shopList = [
```

```
    {shopNo: 100, category: "chicken"},
    {shopNo: 101, category: "pizza"},
    {shopNo: 102, category: "noodle"},
];

const shopAdCampaignList = await Promist.all(shopList.map((shop)
    => AdCampaignAPI.operating(shop.shopNo))
);
// 타입: (AdCampaign | null)[]
// 예시 결과: [AdCampaign, null, AdCampaign]


const shopAds = shopAdCamaignList.filter(NonNullable);
// shopAdCampaignList.filter((value) => NonNullable(value));

// 타입: AdCampaign[]
// 예시 결과: [AdCampaign, AdCampaign]
```

💠 불변 객체 타입으로 활용하기

```
const colors = {
    red: "#F45452",
    green: "#0C952A",
    blue: "#1A7CFF",
};

const getColorHex = (key: string) => colors[key];
// 키 타입을 해당 객체에 존재하는 키값으로 설정하는 게 아니라
// string으로 설정하면 getColorHex 함수의 반환값은 any가 된다.
// -> colors에 어떤 값이 추가될지 모르기 때문이다.
```

getColorHex 함수 인자로 실제 colors 객체에 존재하는 키값만 받도록 설장하자

- `as const`키워드로 객체를 불변 객체로 선언한다.
- `keyof` 연산자를 사용한다.

```
const colors = {
    red: "#F45452",
    green: "#0C952A",
    blue: "#1A7CFF"
} as const;

const getColorHex = (key: keyof typeof colors) => colors[key];
```

➡️ 객체 타입을 구체적으로 설정하면

- 타입에 맞지 않는 값을 전달할 경우 타입 에러가 반환되기 때문에, 컴파일 단계에서 발생할 수 있는 실수를 방지할 수 있다.
- 자동 완성 기능을 통해 객체에 어떤 값이 있는지 쉽게 파악할 수 있다.

Atom 컴포넌트에서 theme style 객체 활용하기

Atom 단위의 작은 컴포넌트`Button, Header, Input` 는 폰트 크기, 폰트 색상, 배경 색상 등 다양한 환경에서 유연하게 사용될 수 있도록 구현되어야 한다.

이러한 설정값은 props로 넘겨주도록 설계한다.

→ 이때 대부분의 프로젝트에서는 해당 프로젝트의 스타일 값을 관리해주는 `theme` 객체를 두고 관리한다.

Atom 컴포넌트에서는 theme 객체의 색상, 폰트 사이즈의 키값을 props로 받은 뒤, theme 객체에서 값을 받아오도록 설계한다.

```
interface Props {
    fontSize?: string;
    backgroundColor?: string;
    color?: string;
    onClick: (event: React.MouseEvent<HTMLButtonElement>) => voi
}

const Button: FC<Props> = ({ fontSize, backgroundColor, color, c
    return (
        <ButtonWrap
            fontSize={fontSize}
            backgroundColor={backgroundColor}
            color={color}
        >
```

```
        {children}
        </ButtonWrap>
    );
};

// Props 인터페이스에서 onClick을 제외한 나머지 속성만 스타일 컴포넌트에서
const ButtonWrap = styled.button<Omit<Props, "onCkick">>`
    color: ${({ color }) => theme.color[color ?? "default"]}; //
    background-color: ${({ backgroundColor }) =>
        theme.bgColor[backgroundColor ?? "default"]};
    font-size: ${({ fontSize }) => theme.fintSize[fontSize ?? "d
`;
```

▼ `Omit`

특정 타입에서 지정된 속성들을 제외한 새로운 타입을 만든다.

```
Omit<Type, Keys>
```

```
interface Props {
    fontSize?: string;
    backgroundColor?: string;
    color?: string;
    onClick: (event: React.MouseEvent<HTMLButtonElement
}

const Button: FC<Props> = ({ fontSize, backgroundColor,
    return (
        <ButtonWrap
            fontSize={fontSize}
            backgroundColor={backgroundColor}
            color={color}
        >
        {children}
        </ButtonWrap>
    );
};

const ButtonWrap = styled.button<Omit<Props, "onCkick">
    color: ${({ color }) => theme.color[color ?? "defau
    background-color: ${({ backgroundColor }) =>
        theme.bgColor[backgroundColor ?? "default"]};
    font-size: ${({ fontSize }) => theme.fintSize[fontS
`;
```

📚 ▼ `??` : 널 병합 연산자

- 오직 null, undefiend만 체크한다.

```
leftExpr ?? rightExpr
// leftExpr이 null이나 undefined면 rightExpr 반환
// 그 외의 경우는 leftExpr 반환
```

❗ fontsize, background 같은 props 타입이 `string` 이면, Button 컴포넌트의 props로 color, background를 넘겨줄 때 키값이 자동 완성되지 않으며 잘못된 키값을 넣어도 에러가 발생하지 않게 된다.

👩 theme 객체로 타입을 구체화해서 해결한다. → `keyof`, `typeof`

타입스크립트 keyof 연산자로 객체의 키값을 타입으로 추출하기

```
interface ColorType {
    red: sting;
    green: string;
    blue: string;
}

type ColorKeyType = keyof ColorType; // 'red' | 'green' | 'blue'
```

타입스크립트 typeof 연산자로 값을 타입으로 다루기

keyof 연산자는 객체 타입을 받는다. 따라서 객체의 키값을 타입으로 다루려면, 값 객체를 타입으로 변환해야 한다.

```
const colors = {
    red: "#F45452",
    green: "#0C952A",
    blue: "#1A7CFF",
}

type ColorsType = typeof colors;
/*
    {
        red: string;
```

```
        green: string;
        blue: string;
    }
*/
```

객체의 타입을 활용해서 컴포넌트 구현하기

> ⛳ keyof, typeof → theme 객체 타입을 구체화하고, string으로 타입을 설정했던 Button 컴포넌트를 개선하자

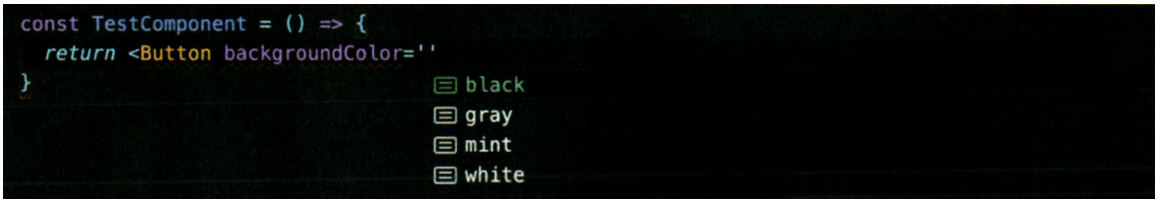
color, backgroundColor, fontSize의 타입을 theme 객체에서 추출하고, 해당 타입을 Button 컴포넌트에 사용한다.

```
import { FC } from "react";
import styped from "styled-components";

const colors = {
    black: "#000000",
    gray: "#22222",
    white: "#FFFFFF",
    mint: "#2AC1BC",
};

const theme = {
    colors: {
        default: colors.gray,
        ...colors
    },
    backgroundColor: {
        default: colors.white,
        gray: colors.gray,
        mint: colors.mint,
        black: colors.black,
    },
    fontSize: {
        default: "16px",
        small: "14px",
        large: "18px",
    },
```

```
};

type ColorType = keyof typeof theme.colors; // type ColorType =
type BackgroundColorType = keyof typeof theme.backgroundColor; /
type FonSizeType = keyof typeof theme.fontSize; // type FonSizeT

interface Props {
    color?: ColorType;
    backgroundColor?: BackgroundColorType;
    fontSize?: FontSizeType;
    onClick: (event: React.MouseEvent<HTMLButtonElement>) => voi
}

const Button: FC<Props> = ({ fontSize, backgroundColor, color, c
    reuturn (
        // ...
    )
}

const ButtonWrap = styled.button<Omit<Props, "onClick">>`
    // ...
`;
```

- 결과

  Button 컴포넌트를 사용하는 곳에서 이제 Button의 매개변수인 backgroundColor 값으로는 BackgroundColorType의 값만 받을 수 있게 되었고, 다른 값을 넣으면 타입 오류가 발생한다.



➡️ theme뿐만 아니라 여러 상숫값을 인자나 props로 받은 다음에, 객체의 키값을 추출한 타입을 활용하면, 객체에 접근할 때 TS의 도움을 받아 실수를 방지할 수 있다.

💠 Record 원시 타입 키 개선하기

객체 선언 시 어떤 값인지 명확하지 않다면 Record의 키를 string or number 같은 원시 타입으로 명시하곤 한다.

❗ TS는 키가 유효하지 않더라도 타입상으로는 문제가 없기 때문에 오류를 표시하지 않는다.
→ 예상치 못한 런타임 에러

무한한 키를 집합으로 가지는 Record

> 무한한 키 집합 : 객체의 키가 무제한으로 존재할 수 있다는 의미이다. 즉, 키 값에 어떤 제약이 없어서 특정한 값들에 한정되지 않고, 모든 값이 가능하다는 뜻이다.

```
// 음식 분류(한식, 일식)를 키로 사용하는 음식 배열이 담긴 객체

type Category = string;
interface Food {
    name: string;
    // ..
}

const foodByCategory: Record<Category, Food[]> = {
    한식: [{name: "제육덮밥"}, {name: "뚝배기 불고기"}],
    일식: [{name: "초밥"}, {name: "텐동"}],
};
```

```
foodByCategory["양식"]; // Food[]로 추론
foodByCategory["양식"].map((food) => console.log(food.name)); //
```

🚨 하지만 `foodByCategory["양식"]` 은 **런타임**에서 undefined가 되어 오류를 반환한다!

👩 JS의 옵셔널 체이닝 등을 사용해 런타임 에러를 방지한다.

💡 `옵셔널 체이닝`

: 객체의 속성을 찾을 때 중간에 `null` 또는 `undefined`가 있어도 오류 없이 안전하게 접근하는 방법

- `?.`
- 중간에 null 또는 undefined 속성이 있는지 검사한다.
- 속성이 존재하면 해당 값을 반환하고, 존재 X으면 undefined를 반환한다.

```
foodByCategory["양식"]?.map((food) => console.log(food.name));
```

❗ 어떤 값이 undefined인지 매번 판단해야 하는 번거로움 / 실수로 undeifined일 수 있는 값을 인지하지 못하고 코드를 작성하면 예상치 못한 런타임 에러 발생

→ 🙆‍♀️ TS의 기능을 활용하여 개발 중에 유효하지 X은 키가 사용되었는지 or undefined일 수 있는 값이 있는지 등을 사전에 파악 가능하다.

유닛 타입으로 변경하기

키가 유한한 집합이라면 유닛타입을 사용할 수 있다.

> 유닛 타입 : 다른 타입으로 쪼개지지 않고 오직 하나의 정확한 값을 가지는 타입

```
// type Category = string;
type Category = "한식" | "일식";

interface Food {
    name: string;
    // ...
}

const foodByCategory: Record<Category, Food[]> = {
    한식: [{name: "제육덮밥"}, {name: "뚝배기 불고기"}],
    일식: [{name: "초밥"}, {name: "텐동"}],
```

```
};

foodByCateogry["양식"]; // 🚨 Property '양식' does not exist on ty
```

❗ 키가 무한해야 하는 상황에는 적합하지 않다.

Partial을 활용하여 정확한 타입 표현하기

👩 키가 무한한 상황에서는 `Partial`을 사용하여 해당 값이 `undefined`일 수 있는 상태임을 표현할 수 있다.

> `Partial`을 사용하면 값이 `undefined`일 수 있다는 의미가 포함된다.

```
type partialRecord<K extends string, T> = Partial<Record<K, T>>;
// [key: string]: Food[] | undefined;
type Category = string;

interface Food {
    name: string;
    // ...
}

const foodByCategory: PartialRecord<Category, Food[]> = {
    한식: [{name: "제육덮밥"}, {name: "뚝배기 불고기"}],
    일식: [{name: "초밥"}, {name: "텐동"}],
};

foodByCateogry["양식"]; // Food[] 또는 undeifined 타입으로 추론
foodByCategory["양식"].map((food) => console.log(food.name)); //
foodCategory["양식"]?.map((food) => console.log(food.name)); // C
```

foodByCategory[key]를 Food[] 또는 undefined로 추론하고, 개발자에게 이 값은 undefined일 수 있으니 해당 값에 대한 처리가 필요하다고 표시해준다.

→ 옵셔널 체이닝, 조건문 ... 등 사전에 조치할 수 있게 되어 예상치 못한 런타임 오류를 줄일 수 있다.