



## 4장 타입 확장하기·좁히기

### ❖ 타입 확장하기

타입 확장 : 기존 타입을 사용해서 새로운 타입을 정의하는 것

- 기본적으로 `interface` 와 `type` 키워드를 사용해서 타입을 정의한다.
- `extends` , 교차 타입 `&` , 유니온 타입 `|` 을 사용하여 타입을 확장한다.

### 타입 확장의 장점

- 중복 제거, 명시적은 코드 작성

```
// <장바구니 기능>
/*
 * 메뉴 요소 타입
 * 메뉴 이름, 이미지, 할인을, 재고 정보
 */
interface BaseMenuItem {
  itemName: string | null;
  itemImageUrl: string | null;
  itemDiscountAmount: number;
  stock: number | null;
}
/*
 * 장바구니 요소 타입
 * 메뉴 타입 + 수량 정보 추가
 */
interface BaseCartItem extends BaseMenuItem {
  quantity: number;
}
```

```
// <장바구니 기능>
type BaseMenuItem = {
  itemName: string
  itemImageUrl: string
  itemDiscountAmount: number
  stock: number | null
}

type BaseCartItem = {
  quantity: number;
} & BaseMenuItem;
```

- 확장성 : 요구사항이 늘어날 때마다 새로운 CartItem 타입 확장하여 정의하기

기존 장바구니 요소에 대한 요구사항이 변경되어도 BaseCartItem 타입만 수정하면 된다.

```

/*
 * 수정할 수 있는 장바구니 요소 타입
 * 품질 여부, 수정할 수 있는 옵션 배열 정보 추가
 */
interface EditableCartItem extends BaseCartItem {
    isSoldOut: boolean;
    optionGroups: SelectableOptionGroup[];
}
/*
 * 이벤트 장바구니 요소 타입
 * 주문 가능 여부에 대한 정보 추가
 */
interface EventCartItem extends BaseCartItem {
    orderable: boolean;
}

```

## 유니온 타입

= 합집합

- 유니온으로 선언된 값은 유니온 타입에 포함된 모든 타입이 공통으로 갖고 있는 속성에만 접근 가능하다!

```

interface CookinStep {
    orderId: string;
    price: number;
}

interface DeliveryStep {
    orderId: string;
    time: number;
    distance: string;
}

function getDeliveryDistance(step: CookingStep | DeliveryStep) {
    return step.distance;
    // 🚨 distance는 DeliveryStep에만 존재하여 에러 발생
}

```

step이라는 유니온 타입은 Cooking 또는 DeliveryStep 타입에 해당할 뿐이지 CookingStep 이면서 DeliveryStep인 것은 아니다.

## 교차 타입

= 모두 만족(교집합)하는 값의 타입(집합)

기존 타입을 합쳐 필요한 모든 기능을 가진 하나의 타입을 만드는 것

- (두 인터페이스의 속성이 겹치지 않는 경우) : 여기서 교차 타입은 "두 타입의 모든 속성을 포함해야 한다"는 의미이다. 즉, 두 타입의 "요구사항을 모두 만족"하는 타입을 만드는 것

```
type BaedalProgress = CookingStep & DeliveryStep;

function logBaedalInfo(progress: BaedalProgress) {
  console.log(`주문 금액: ${progress.price}`);
  console.log(`배달 거리: ${progress.distance}`);
}

// ✅ 즉, BaedalProgress 타입의 progress 값은 CookingStep의 price &
// DeliveryStep의 distance 속성 모두 포함하고 있다.
```

🌟 **유니온 타입과 차이점** : BaedalProgress는 CookingStep과 DeliveryStep 타입을 합쳐 모든 속성을 가진 단일 타입이 된다.

- 교차 타입이므로 두 타입을 모두 만족하는 경우에만 유지된다.

```
type IdType = string | number;
type Numeric = number | boolean;

type Universal = IdType & Numeric;

// ✅ number이면서 number인 경우
```

## extends와 교차 타입

`extends` 를 사용해서 교차타입을 작성할 수도 있다.

🌟 유니온타입과 교차타입을 사용한 새로운 타입은 오직 `type` 으로만 선언할 수 있다.

```
// 1. interface 내부에서 유니온 타입 사용 가능
interface User {
  id: string | number; // ✅ 유니온 타입 사용 가능
  type: 'admin' | 'user'; // ✅ 유니온 타입 사용 가능
  info: UserInfo & LogInfo; // ✅ 교차 타입 사용 가능
}

// 2. interface 자체를 유니온/교차로 선언은 불가능
interface CombinedInterface = User | Admin; // ❌ 에러
interface CombinedInterface = User & Admin; // ❌ 에러

// type으로는 가능
type CombinedType = User | Admin; // ✅ 가능
type IntersectionType = User & Admin; // ✅ 가능
```

- `extends` 를 사용한 타입 ≠ 교차타입 100% 상응 X

```
interface DeliveryTip {
  tip: number;
}

interface Filter extends DeliveryTip {
  tip: string; // 🚨
  // Interface 'Filter' ioncorrec
  // Types of property 'tip' are
  // Type 'string' is not assign
}
```

```
type DeliveryTip = {
  tip: number;
}

type Filter = DeliveryTip & {
  tip: string;
}
```

`type` 은 교차 타입으로 선언되었을 때 새롭게 추가되는 속성에 대해 미리 알 수 없기 때문에 선언 시 에러가 발생하지 X한다. 하지만 tip이라는 같은 속성에 대해 서로 호환되지 않는 타입이 선언되어 결국 `never` 타입이 된다.

## 배달의민족 메뉴 시스템에 타입 확장 적용하기



1인분



족발·보쌈



찜·탕·찌개



돈까스·화일식



피자

배달의민족 배달 서비스의 메뉴 목록

```

/*
 * 메뉴에 대한 타입
 * 메뉴 이름, 메뉴 이미지 정보
 */
interface Menu {
  name: string;
  image: string;
}

```

```

function MainMenu() {
  // Menu 타입을 원소로 갖는 배열
  const menuList: Menu[] = [{name: "1인분", image: "1인분.png"}],

  return (
    // jsx
  )
}

```

특정 메뉴의 중요도를 다르게 주기 위한 요구사항이 추가되었다.

1. 특정 메뉴를 길게 누르면 gif파일이 재생되어야 한다.
2. 특정 메뉴는 이미지 대신 별도의 텍스트만 노출되어야 한다.

• 방법1, 방법2

```

/*
 * 방법 1 : 타입 내에서 속성 추가
 * 기존 Menu 인터페이스에 추가된 정보를 전부 추가하기
 */
interface Menu {
  name: string;

```

```

    image: string;
    gif?: string; // 요구사항1
    text?: string; // 요구사항2
}

/**
 * 방법 2 : 기존 Menu 인터페이스는 유지한 채, 각
 * 각 요구사항에 따른 별도 타입을 만들어 확장시키는 구조
 */
interface Menu {
    name: string;
    image: string;
}

// gif를 활용한 메뉴 타입 : Menu인터페이스를 확장해서 반드시 gif 값을 갖도록
interface SpecialMenu extends Menu {
    gif: string; // 요구사항1
}

// 별도의 텍스트를 활용한 메뉴 타입 : Menu인터페이스를 확장해서 반드시 text
interface PackageMenu extends Menu {
    text: string; // 요구사항2
}

```

- 3가지 종류의 메뉴 목록이 있을 때 각 방법을 적용해보자!

```

/**
 * 각 배열은 서버에서 받은 응답 값이라고 가정
 */
const menuList = [
    { name: "짬", image: "짬.png" },
    { name: "찌개", image: "찌개.png" },
    { name: "회", image: "회.png" },
];

const specialMenuList = [
    { name: "돈까스", image: "돈까스.png", gif: "돈까스.gif" },
    { name: "피자", image: "피자.png", gif: "피자.gif" },
];

const packageMenuList = [
    { name: "1인분", image: "1인분.png", text: "1인 가구 맞춤형" },
    { name: "족발", image: "족발.png", text: "오늘은 족발로 결정" },
];

```

## 방법 1: 하나의 타입에 여러 속성을 추가할 때

```
menuList: Menu[]
specialMenuList: Menu[]
packageMenuList: Menu[]

specialMenuList.map((menu) => menu.text); // 🚨 TypeError: Cannot
```

specialMenuList는 `Menu` 타입의 원소를 갖기 때문에 `text` 속성에도 접근할 수 있다.

하지만 specialMenuList 배열의 모든 원소는 `text` 속성을 가지고 있지 않다.

## 방법 2: 타입을 확장하는 방식

```
menuList: Menu[]

specialMenuList: Menu[] // 🚨
specialMenuList: SpecialMenu[]

packageMenuList: Menu[] // 🚨
packageMenuList: PackageMenu[]

specialMenuList.map((menu) => menu.text); // 🚨 Property 'text' d
```

이때는 프로그램을 실행하지 않고도 타입이 잘못되었음을 알 수 있다.



주어진 타입에 무분별하게 속성을 추가하여 사용하는 것 < 타입을 확장해서 사용하는 것이 좋다.

- 적절한 네이밍을 사용해서 타입의 의도를 명확히 표현
- 코드 작성 단계에서 예기치 못한 버그 예방

## 🔗 타입 좁히기 - 타입 가드

### 타입 가드에 따라 분기 처리하기

**분기 처리** : 조건문과 타입 가드를 활용하여 변수나 표현식의 타입 범위를 좁혀 다양한 상황에 따라 다른 동작을 수행하는 것

**타입 가드** : 런타임에 조건문을 사용하여 타입을 검사하고 타입 범위를 좁혀주는 기능

- JS 연산자를 사용한 타입 가드 : `typeof`, `instanceof`, `in`
- 사용자 정의 타입 가드

**스코프** : 변수와 함수 등의 식별자가 유효한 범위. 즉, 변수와 함수를 선언하거나 사용할 수 있는 영역.

- ex) 어떤 함수가 `A | B` 타입의 매개변수를 받는다. 인자 타입이 `A` or `B`일 때를 구분해서 로직을 처리하고 싶다면?

! 컴파일 시 타입 정보는 모두 제거되어 런타임에 존재하지 않기 때문에 타입을 사용하여 조건 `if문` 을 만들 수는 X다.

👤 컴파일해도 타입 정보가 사라지지 않는 방법 → **타입 가드**

## 원시 타입을 추론할 때: **typeof** 연산자 활용하기



`typeof` 연산자를 사용하여 검사할 수 있는 타입 목록

- `string`
- `number`
- `boolean`
- `undefined`
- `object`
- `function`
- `bigint`
- `symbol`

```
// 매개변수의 타입 : string or Date
// 반환값의 타입 : string or Date
```



```
// 화살표 함수 구현
const replaceHyphen: (date: string | Date) => string | Date = (date) => {
  if (typeof date === "string") {
    // 이 분기에서는 date 타입이 string으로 추론된다.
    return date.replace(/-/g, "/");
  }

  return date;
};
```



### 함수 정의 문법 정리

#### 1. 함수 선언식

```
function replaceHyphen(date: string | Date): string | Date {
  // ...
}
```

#### 2. 화살표함수 + 타입 표기

```
const replaceHyphen = (date: string | Date): string | Date => {
  // ...
};
```

#### 3. 화살표함수 + 전체 타입 정의

- 변수 `replaceHyphen` 에 전체 함수 타입을 먼저 정의
- 실제 매개변수 `date` 에는 타입을 명시하지 않음 (타입이 추론됨)
  - `date`의 타입이 `ReplaceHyphenType`으로부터 추론됨

```
const replaceHyphen: (date: string | Date) => string | Date = (date) => {
  // ...
};
```

## 인스턴스화된 객체 타입을 판별할 때: instanceof 연산자 활용하기

- `typeof` 는 주로 원시 타입을 판별하는 데 사용
- `instanceof` 는 인스턴스화된 객체 타입을 판별하는 타입 가드로 사용

```
A instanceof B
// A : 타입을 검사할 대상 변수
// B : 특정 객체의 생성자
// A의 프로토타입 체인에 생성자 B가 존재하는지를 검사해서 존재 시, true 없을
```

```
interface Range {
  start: Date;
  end: Date;
}

interface DatePickerProps {
  selectedDates?: Date | Range;
}

// DatePicker는 화살표함수가 아니라 컴포넌트이다. 일반적으로 컴포넌트에서는
const DatePicker = ({ selectedDates }: DatePickerProps) => {
  const [selected, setSelected] = useState(convertToRange(selectedDates));
  // ...
};

// selected 매개변수가 Date인지 검사한 후에 Range 타입의 객체를 반환할 수
export function convertToRange(selected?: Date | Range): Range {
  return selected instanceof Date
    ? {start: selected, end: selected}
    : selected;
}
```

```
// <HTMLInputElement>에 존재하는 blur 메서드 사용을 위한 HTMLInputElement
const onKeyDown = (event: React.KeyboardEvent) => {
  if (event.target instanceof HTMLInputElement && event.key === 'enter') {
    // 이 분기에서는 event.target의 타입이 HTMLInputElement이며
    // event.key가 'enter'이다
    event.target.blur();
    onCTAButtonClick(event);
  }
}
```

```
}  
};
```

## 객체의 속성이 있는지 없는지에 따른 구분 : **in** 연산자 활용하기

**in** : 객체에 속성이 있는지 확인한 다음에 true 또는 false를 반환한다.

→ 여러 객체 타입을 유니온 타입으로 가지고 있을 때 **in** 을 사용해서 속성의 유무에 따라 조건 분기 할 수 있다.

```
A in B  
// A라는 속성이 B 객체에 존재하는지를 검사한다.
```

- 프로토타입 체인으로 접근할 수 있는 속성이면 전부 true를 반환한다.
- B 객체에 존재하는 A 속성에 undefined를 할당한다고 해서 false를 반환하지 않는다.  
delete 연산자를 사용하여 객체 내부에서 해당 속성을 제거해야지 false를 반환한다.
- JS의 **in** 은 런타임의 값만을 검사하지만, TS에서는 객체 타입에 속성이 존재하는지를 검사한다.

```
interface BasicNoticeDialogProps {  
  noticeTitle: string;  
  noticeBody: string;  
}  
  
interface NoticeDialogWithCookieProps extends BasicNoticeDialogF  
  cookieKey: string;  
  noForADay?: boolean;  
  neverAgain?: boolean;  
}  
  
export type NoticeDialogProps =  
  | BasicNoticeDialogProps  
  | NoticeDialogWithCookieProps; // BasicNoticeDialogProps를 상속
```

NoticeDialog 컴포넌트가 props로 받는 객체 타입이  
BasicNoticeDialogProps인지 NoticeDialogWithCookieProps인지  
에 따라 렌더링하는 컴포넌트가 달라지도록 해보자.

```
// cookieKey 속성을 가졌는지 아닌지에 따라 in연산자로 조건을 만들면 된다.
const NoticeDialog: Reqct.FC<NoticeDialogProps> = (props) => {
  // 얼리 리턴
  if ("cookieKey" in props) return <NoticeDialogWithCookie {...props} />;
  return <NoticeDialogBase {...props} />;
};
```

얼리 리턴 : 특정 조건에 부합하지 않으면 바로 반환하는 것을 말한다.

## is 연산자로 사용자 정의 타입 가드 만들어 활용하기

반환 타입이 타입 명제인 함수를 정의하여 사용할 수 있다.

**타입 명제** : 함수의 반환 타입에 대한 타입 가드를 수행하기 위해 사용되는 특별한 형태의 함수

```
A is B
// A : 매개변수 이름
// B : 타입
```

- 참/거짓을 반환하면서 반환 타입을 타입 명제로 지정하게 되면, 반환값이 참일 때 A 매개변수의 타입을 B 타입으로 취급하게 된다.

```
// 타입가드 함수
// string 타입의 매개변수가 destinationCodeList 배열의 원소 중 하나인지
const isDestinationCode = (x: string): x is DestinationCode =>
  destinationCodeList.includes(x);
```



▼ `includes()` : 배열에서 특정 값이 포함되어 있는지 확인할 수 있는 JavaScript 배열 메서드

```
const destinationCodeList = ['ICN', 'LAX', 'JFK', 'LHR']

// includes 사용
console.log(destinationCodeList.includes('ICN')); // true
console.log(destinationCodeList.includes('ABC')); // false
```

✓ 반환값을 `boolean` 이 아닌 `x is DestinationCode` 로 타이핑 해야 하는 이유

Typescript는 `x str` 가 `destinationCodeList` 중 하나인 것은 확실한데 여전히 `string`으로 본다. 그래서 타입가드를 사용하여 Typescript에게 `DestinationCode` 타입임을 알려준다.

```
// 예제 이해를 위한 추가 //
const destinationCodeList = ['ICN', 'LAX'] as const;
type DestinationCode = typeof destinationCodeList[number]; // 'ICN' | 'LAX'

const getAvailableDestinationNameList = async (): Promise<DestinationNameSet> => {
  const data = await axiosRequest<string[]>("get", ".../destinationCodes");
  const destinationNames: DestinationName[] = [];
  data?.forEach((str) => {
    if (isDestinationCode(str)) { // data의 str이 DestinationCode 타입인지 확인
      destinationNames.push(DestinationNameSet[str]);
    }
    // ⚠ isDestinationCode의 반환값에 is를 사용하지 않고, boolean만 반환
    // TS는 str이 여전히 string 타입이라고 생각한다!
    // DestinationNameSet의 인덱스로 일반 string을 사용할 수 없다.
  });
  return destinationNames;
}
```



## 타입 좁히기 - 식별할 수 있는 유니온

식별할 수 있는 유니온 은 타입 좁히기에 널리 사용되는 방식이다.

= 태그된 유니온

## 에러 정의하기

```
// 배달의민족의 서비스 - 선물을 보낼 때 필요한 값을 사용자가 올바르게 입력했는지
// 에러 종류 : 텍스트 에러, 토스트 에러, 얼럿 에러
type TextError = {
  errorCode: string;
  errorMessage: string;
};
type ToastError = {
  errorCode: string;
  errorMessage: string;
  toastShowDuration: number; // 토스트를 띄워줄 시간
};
type AlertError = {
  errorCode: string;
  errorMessage: string;
  onConfirm: () => void; // 얼럿 창이 확인 버튼을 누른 뒤 액션
}
```

유니온 타입인 `ErrorFeedbackType`의 원소를 갖는 배열 `errorArr`를 정의함으로써 다양한 에러 객체를 관리할 수 있게 된다.

```
type ErrorFeedbackType = TextError | ToastError | AlertError;
const errorArr: ErrorFeedbackType[] = [
  // ErrorFeedbackType[]은 배열의 각 요소가 TextError | ToastError | AlertError
  { errorCode: "100", errorMessage: "텍스트 에러" },
  { errorCode: "200", errorMessage: "토스트 에러", toastShowDuration: 3000 },
  { errorCode: "300", errorMessage: "얼럿 에러", onConfirm: () => {} }
]
```

! `ToastError`의 `toastShowDuration` 필드와 `AlertError`의 `onConfirm` 필드를 모두 가지는 객체에 대해서는 타입 에러를 뱉어야 하지만, JS는 덕 타이핑 언어이기 때문에 별도의 타입 에러를 뱉지 않는다.

JavaScript는 객체가 필요한 속성만 가지고 있으면 타입과 관계없이 동작한다. `덕 타이핑`

```
const errorArr: ErrorFeedbackType[] = [
  // ...
  {
    errorCode: "999",
    errorMessage: "잘못된 에러",
    toastShowDuration: 3000,
    onConfirm: () => {},
  }, // 🚨 에러 발생
]
```

➡ 의미를 알 수 없는 무수한 에러 객체가 생겨날 위험성이 커진다.

## 식별할 수 있는 유니온

👤 에러 타입 구분 방법이 필요하다. → 식별할 수 있는 유니온

**식별할 수 있는 유니온** : 타입 간의 구조 호환을 마기 위해 타입마다 구분할 수 있는 판별자를 달아주어 포함 관계를 제거하는 것

```
// 판별자 정의하기 : errorType 필드
// 각 에러 타입마다 이 필드에 대해 다른 값을 가지도록 하여 판별자를 달아주면
// 이들은 포함 관계를 벗어나게 된다.
type TextError = {
  errorType: "TEXT",
  errorCode: string;
  errorMessage: string;
};
type ToastError = {
  errorType: "TOAST";
  errorCode: string;
  errorMessage: string;
  toastShowDuration: number;
};
type AlertError = {
  errorType: "ALERT";
  errorCode: string;
  errorMessage: string;
  onConfirm: () => void;
}
```



판별자를 사용하면 좋은 이유

- 더 명확한 타입 에러 메시지

```
const errorArr: ErrorFeedbackType[] = [
  // 1 errorType(판별자) 없을 때
  {
    errorCode: "999",
    errorMessage: "잘못된 에러",
    toastShowDuration: 3000,
    onConfirm: () => {}
  }
  // ✗ 에러 메시지:
  // "Type '{ errorCode: string; errorMessage: string; toastShowDuration: number; onConfirm: () => void; }'
  // is not assignable to type 'ErrorFeedbackType'"
  // -> "이 객체는 ToastError도 아니고, TextError도 0

  // 2 errorType(판별자) 있을 때
  {
    errorType: "TEXT",           // TEXT 타입이라고 명
    errorCode: "999",
    errorMessage: "잘못된 에러",
    toastShowDuration: 3000      // ✗ "'toastShowDu
    onConfirm: () => {}         // ✗ "'onConfirm'

  }
];
```

- 타입 좁히기가 더 안전하다.

```
function handleError(error: ErrorFeedbackType) {
  // errorType 없을 때
  if ('toastShowDuration' in error) {
    // 속성 체크로 타입 좁히기
    // 실수로 다른 타입에도 toastShowDuration을 추가하면
  }

  // errorType 있을 때
  if (error.errorType === "TOAST") {
    // 명시적인 타입 체크
  }
}
```



```
// 실수할 여지가 적음
    }
}
```

## 식별할 수 있는 유니온의 판별자 선정

식별할 수 있는 유니온의 판별자는 유닛 타입으로 선언되어야 정상 동작한다.

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

ex) null, undefined, 리터럴 타입, true, 1 등 정확한 값을 나타내는 타입


ex) 유닛타입으로 적용 안되는 타입 : 다양한 타입 할당이 가능한 void, string, number ..

## Exhaustiveness Checking으로 정확한 타입 분기 유지하기 (feat. never)

**Exhaustiveness Checking** : 모든 케이스에 대해 철저하게 타입을 검사하는 것. 타입 좁히기에 사용되는 패러다임 중 하나.

- 모든 케이스에 대한 타입 분기 처리를 해주지 X했을 때, 컴파일타임 에러 발생하게 하는 것

! 타입 가드를 사용해서 타입에 대한 분기 처리를 수행하면 되지만, 때로는 모든 케이스에 대한 분기 처리를 해야만 유지보수 측면에서 안전하다고 생각되는 상황이 생긴다.

 **Exhaustiveness Checking** 을 통해 모든 케이스에 대한 타입 검사를 강제한다.

### ex) 상품권

```
// 배달의민족 선물하기 서비스 - 다양한 상품권
// <상품권 가격에 따라 상품권 이름을 반환해주는 함수>
```

```
type ProductPrice = "10000" | "20000";
```

```
const getProductName = (productPrice: ProductPrice): string => {
  if (productPrice === "10000") return "배민상품권 1만 원";
  if (productPrice === "20000") return "배민상품권 2만 원";
  else {
    return "배민상품권";
  }
}
```

새로운 상품권이 생겨서 `ProductPrice` 타입이 업데이트되어야 한다.

```
type ProductPrice = "10000" | "20000" | "5000";

const getProductName = (productPrice: ProductPrice): string => {
  if (productPrice === "10000") return "배민상품권 1만 원";
  if (productPrice === "20000") return "배민상품권 2만 원";
  if (productPrice === "5000") return "배민상품권 5천 원"; // 조건
  else {
    return "배민상품권";
  }
};
```

! `ProductPrice` 타입이 업데이트되었을 때 `getProductName` 함수도 함께 업데이트되어야 한다.

→ 그러나 `getProductName` 함수를 수정하지 않아도 별도 에러가 발생하는 것은 아니기 때문에 실수할 여지가 있다.

👤 모든 타입에 대한 타입 검사를 강제하자.

```
type ProductPrice = "10000" | "20000" | "5000";

const getProductName = (productPrice: ProductPrice): string => {
  if (productPrice === "10000") return "배민상품권 1만 원";
  if (productPrice === "20000") return "배민상품권 2만 원";
  // if (productPrice === "5000") return "배민상품권 5천 원";
  else { // "5000" 타입
    exhaustiveCheck(productPrice); // 🚨 Error: Argument of
    return "배민상품권";
  }
};
```

```
const exhaustiveCheck = (param: never) => {
  // "5000" 타입은 never 타입에 할당할 수 없다!
  throw new Error("type error!");
}
```

ProductPrice 타입 중 5000이라는 값에 대한 분기 처리를 하지 않아서 즉, 철저하게 검사하지 않아서 발생한 에러이다.

**never** : "절대로 발생할 수 없는 값"의 타입

즉,

**never** 타입의 매개변수에는 어떤 값도 할당할 수 없다.

## 유형 이야기 : never

- **never** : 어떤 값이든 never 그 자체를 제외하고는 never로 할당할 수 없다 → 이 개념 하나로 다양한 패턴을 만들어 타입을 잡아주니 참 신기한 타입
- 런타임 코드에 **exhaustiveCheck** 가 포함되며 프로덕션 코드와 테스트 코드가 같이 섞여 있는 듯한 느낌

프로덕션 코드에 어서션을 추가하는 것도 하나의 패턴이다.

- 단위 테스트의 어서션 : 특정 단위의 결과를 확인하는 느낌
- 코드상의 어서션 : 코드 중간중간에 무조건 특정 값을 가지는 상황을 확인하기 위한 디버깅 or 주석 같은 느낌

어서션 > 아무 주석을 단다거나 문서화가 안 되어 있는 것

**어서션** : 타입 단언 ex) **as**