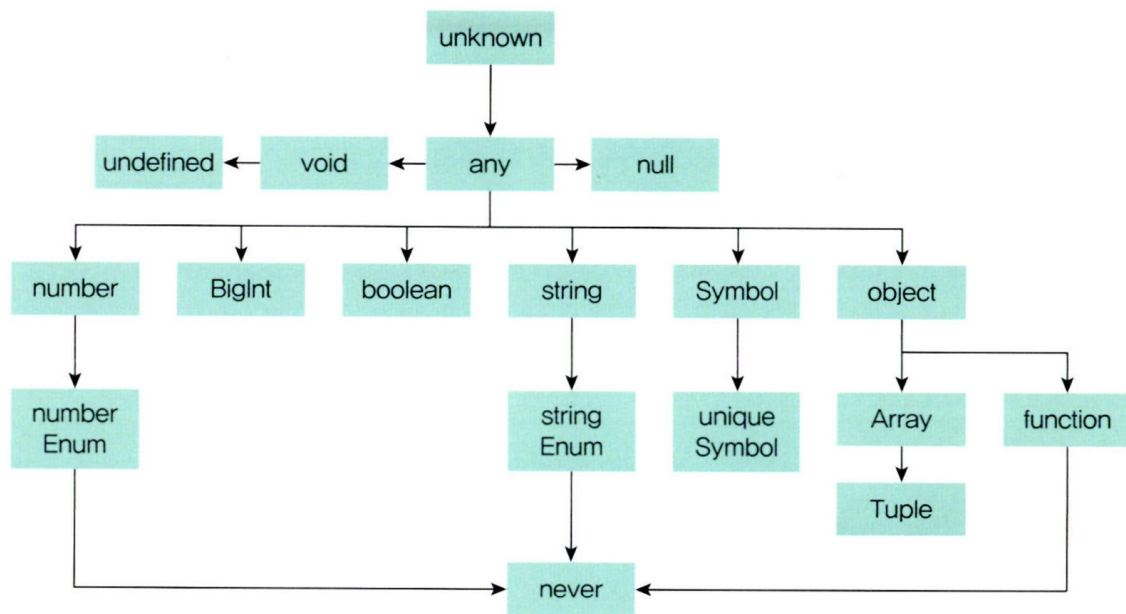




3장 고급타입

❖ TS만의 독자적 타입 시스템



▶ 타입스크립트의 타입 계층 구조

any 타입

JS에 존재하는 모든 값을 오류 없이 받을 수 있다. 즉, 타입을 명시하지 않은 것과 같다.

```
let state: any;

state = { value: 0 }; // 객체 할당
state = 100; // 숫자 할당
state = "hello world"; // 문자열 할당
state.foo.bar = () => console.log("this is any type"); // 심지어
```



any 타입을 변수에 할당하는 것은 지양해야 할 패턴이다.

- tsconfig.json 파일에서 `noImplicitAny` 옵션 활성화
타입이 명시되지 않은 변수의 암묵적인 any 타입에 대한 경고를 발생시킨다.

✓ TS에서 any 타입을 어쩔 수 없이 사용해야 하는 대표적인 3가지 사례

1. 개발 단계에서 임시로 값을 지정해야 할 때

매우 복잡한 구성 요소로 이루어진 개발 과정에서 추후 값이 변경될 가능성이 있거나 아직 세부 항목에 대한 타입이 확정되지 않은 경우

2. 어떤 값을 받아올지 or 넘겨줄지 정할 수 없을 때

ex) API 요청 및 응답 처리, 콜백함수 전달, 타입이 잘 정제되지 않아 파악 힘든 외부 라이브러리 등을 사용할 때

→ 주고받을 값이 명확하지 않을 때 열린타입 `any` 을 선언해야 할 수 있다.

```
// 피드백을 나타내기 위해 모달 창을 그릴 때 사용되는 인자를 나타내는 타입
type FeedbackModalParams = {
  show: boolean;
  content: string;
  cancelButtonText?: string;
  confirmButtonText?: string;
  beforeOnClose?: () => void;
  action?: any; // 모달 창을 그릴 때 실행될 함수
}
```

모달 창을 화면에 그릴 때 다양한 범주의 액션에 따라 인자의 개수나 타입을 일일이 명시하기 힘들 수 있다. `any`

3. 값을 예측할 수 없을 때 암묵적으로 사용

외부 라이브러리나 웹 API의 요청에 따라 다양한 값 반환하는 API ex) Fetch API

- Fetch API의 일부 메서드는 요청 이후의 응답을 특정 포맷으로 파싱하는데 이때 반환 타입이 `any` 로 매핑되어 있다.



Fetch API와 Rest API

Fetch API : 비동기 네트워크 통신을 구현하기 위해 클라이언트 측에서 브라우저에 제공하는 Web API

Rest API : 웹 서비스를 설계하는 아키텍처 스타일/방식

➡ Fetch API는 REST API를 호출하는데 사용되는 도구이다.

```

asnc function load() {
  const response = await fetch("https://api.com");
  const data = await response.json(); // response.json()의
  // Response 본문을 JSON으로 파싱하는 작업도 비동기이기 때문에 await

  return data;
}

```

unknown 타입

any 타입과 유사하게 모든 타입의 값이 할당될 수 있다. 그러나 any를 제외한 다른 타입으로 선언된 변수에는 unknown 타입 값을 할당할 수 X다.

✓ any 타입 vs. unknown 타입 비교

any	unknown
<ul style="list-style-type: none"> - 어떤 타입이든 any 타입에 할당 가능 - any 타입은 어떤 타입으로도 할당 가능(단 never는 제외) 	<ul style="list-style-type: none"> - 어떤 타입이든 unknown 타입에 할당 가능 - unknown 타입은 any 타입 외에 다른 타입으로 할당 불가능

```
let unknownValue: unknown;
```

```
unknownValue = 100; // 숫자
```

```
unknownValue = "hello world"; // 문자열
```

```
unknownValue = () => ("this is any type") // 함수
```

```
let someValue1: any = unknownValue; // ✓ any 타입으로 선언된 변수를
```

```
let someValue2: number = unknownValue; // ❌  
let someValue3: string = unknownValue; // ❌
```

```
// any 사용시  
let valueAny: any = "hello";  
valueAny.toLowerCase(); // 문제없이 실행됨  
valueAny = 42;  
valueAny.toLowerCase(); // 런타임 에러!  
  
// unknown 사용시  
let valueUnknown: unknown = "hello";  
valueUnknown.toLowerCase(); // 컴파일 에러! ❌  
// Object is of type 'unknown'  
  
// unknown은 타입 체크 후 사용해야 함  
if (typeof valueUnknown === "string") {  
    valueUnknown.toLowerCase(); // 정상 작동! ✅  
}
```



any타입과 비슷한 unknown 타입이 추가된 이유

ex) unknown 타입 변수에 할당할 때는 컴파일러가 경고 주지 않지만, 실행하면 에러 발생한다.

❗ 함수뿐만 아니라 객체의 속성 접근, 클래스 생성자 호출을 통한 인스턴스 생성 등 객체 내부에 접근하는 모든 시도에서 에러 발생



```
// 할당하는 시점에서는 에러가 발생하지 않음
const unknownFunction: unknown = () => console.log("this is unknown type");

// 하지만 실행 시에는 에러가 발생; Error: Object is of type 'unknown'.ts (2571)
unknownFunction();
```



unknown 타입으로 선언된 변수는 값을 가져오거나 내부 속성에 접근할 수 없다.

unknown 타입은 any 타입을 특정 타입으로 수정해야 하는 것을 깜빡하고 누락하는 상황을 보완하기 위해 등장한 타입이다.

🌟 any 타입과 유사하지만 타입검사를 강제하고, 타입이 식별된 후에 사용할 수 있기 때문에 any 타입보다 더 안전하다. → 그래서 unknown 타입 대체 방법이 권장된다!



우아한에서 unkown 사용법 🚚

- 강제 타입 캐스팅을 통해 타입을 전환할 때 사용한다.

강제 타입 캐스팅 : 이 값을 다른 타입으로 강제로 취급하겠다. **as** 키워드를 사용한다.



▼ 강제 타입 캐스팅을 사용해야 하는 경우

1. TS가 타입을 추론하지 못할 때

```
// DOM 요소 접근 시
const myCanvas = document.getElementById('main_ca
```

2. 외부 라이브러리나 API 응답 처리할 때

```
interface UserData {
  id: number;
  name: string;
}

// API 응답을 특정 타입으로 변환
const response = await fetch('/api/user');
const userData = await response.json() as UserDat
```

3. 복잡한 타입 변환이 필요할 때

```
interface ComplexType {
  data: {
    users: {
      id: number;
      name: string;
    }[];
  };
}

const data = JSON.parse(someString) as ComplexTyp
```

4. environment variables 환경변수 타입 지정할 때

Node.js의 기본 process.env 환경변수 객체 와 우리가 실제로 사용할 환경변수의 타입이 다르기 때문에 형변환이 필요하다.

- 기본 process.env의 타입은 string | undefined 이다.

```
// unknown을 통한 이중 캐스팅
const env = process.env as unknown as ProcessEnv;
```

```
// unknown은 모든 타입으로 변환될 수 있는 타입이므로 A->U
```

5. 라이브러리 타입 정의가 불안정할 때

```
// 라이브러리 함수의 반환 타입이 너무 넓게 정의된 경우  
const result = someLibraryFunction() as SpecificT
```

- `any` 는 무엇이든 괜찮다, `unknown` 은 뭔지 모르지만 하나씩 테스트하면서 뭔지 알아내보자.
- 예상할 수 없는 데이터라면 `unknown` 을 쓴다.
 - TS 4.4부터 try-catch 에러의 타입이 `any`에서 `unknown`으로 변경되어서 에러 핸들링할 때도 `unknown` 사용한다.

void 타입

함수에 전달되는 매개변수 타입과 반환하는 타입을 지정해야 하는데, 아무런 값을 반환하지 않는 경우에 `void` 타입을 매핑한다.

- 함수에서 명시적인 반환문을 작성하지 않으면, JS에서는 `Undefined` 를 반환하고 TS에서는 `void` 타입이 사용된다.

```
function showModal(type: ModalType): void {  
    feedbackSlice.actions.createModal(type);  
}  
  
// 화살표 함수로 작성 시  
const showModal = (type: ModalType) : void => {  
    feedbackSlice.actions.createModal(type);  
}
```



void 타입으로 지정된 변수는 undefined 또는 null 값만 할당할 수 있다.

- null 값을 할당할 수 없는 경우
 - tsconfig.json에서 `strictNull-Checks` 옵션이 설정된 경우
 - 컴파일 시 해당 플래그 설정이 실행된 경우

```
let voidValue: void = undefined;

// strictNullChecks 비활성화된 경우에 가능
voidValue = null;
```



일반적으로 함수 자체를 다른 함수의 인자로 전달하는 경우가 아니라면 void 타입은 잘 명시하지 않는 경향이 있다. TS 컴파일러가 알아서 함수 타입을 void로 추론해주기 때문이다.

never 타입

값을 반환할 수 없는 타입. 함수와 관련하여 많이 사용되는 타입.

- 모든 타입의 하위 타입. 따라서 `any` 타입이라 할지라도 never 타입에 할당될 수 X다.

✓ JS에서 값을 반환할 수 없는 예 2가지

1. 에러를 던지는 경우

JS에서는 런타임에 의도적으로 에러를 발생시키고 캐치할 수 있다.

`throw` 키워드를 사용하여 에러를 발생시키는데, 이때 값을 반환하지 않는 것으로 간주한다.

- 특정 함수가 실행 중 마지막에 에러를 던지는 작업을 수행하면, 해당 함수의 반환 타입은 never이다.

```
function generateError(res: Response): never {
  throw new Error(res.getMessage());
}
```

2. 무한히 함수가 실행되는 경우

드물게 함수 내에서 무한 루프를 실행하는 경우가 있을 수 있다. (무한루프 = 함수가 종료되지 X음)


```
function checkStatus(): never {
  while (true) {
    // ...
  }
}
```

Array 타입



`Object.prototype.toString.call(...)` : 객체의 타입을 알아내는 데 사용하는 함수

➤ `typeof`를 사용하지 않고 이 함수를 사용하는 이유 : `typeof`는 단순히 `object` 타입의 알려주지만, 이 함수는 `객체의 인스턴스` 까지 알려준다.

```
const arr = [];
console.log(Object.prototype.toString.call(arr)) // '[Obje
```

- JS를 제외하고 TS, Java, C++ 등 다른 정적 언어에서는 배열의 원소로 하나의 타입만 사용하도록 명시한다.

대개 정적 타입의 언어에서는 배열을 선언할 때 크기까지 동시에 제한하지만, TS에서는 배열의 크기까지 제한하지 않는다.

```
// JS
const fn = () => console.log(1);
const array = [1, "string", fn];
```

```
// TS
// 1번째 선언 방법
const array: number[] = [1, 2, 3] // 숫자에 해당하는 원소만 허용

// 2번째 선언 방법
const array: Array<number> = [1, 2, 3];
```

✓ 여러 타입을 모두 관리해야 하는 배열 선언 방법 `유니온 타입 사용`

```
const array1: Array<number | string> = [1, "string"];

const array2: number[] | string[] = [1, "string"];
const array3: (number | string)[] = [1, "string"];
```

튜플

배열 타입의 하위 타입.

배열과 달리 튜플은 기존 TS의 배열 기능에 **길이 제한**까지 추가한 타입 시스템.

- 배열의 특정 인덱스에 정해진 타입을 선언한다.

```
let tuple: [number] = [1];
tuple = [1, 2]; // 불가능
tuple = [1, "string"]; // 불가능

let tuple: [number, string, boolean] = [1, "string", true]; // C
```

✓ 튜플의 쓰임새

ex) `useState` 는 튜플 타입을 반환한다. + 구조분해할당

ex) 객체에 적용한 객체 구조분해할당

| 객체 구조분해할당에서 이름 바꾸기

```
const useStateWithObject = (initialValue: any) => {
  ...
  return {value, setValue};
}

const {value, setValue} = useStateWithObject(false); // 해당 함수(
// value와 setValue라는 원래 이름 그대로 사용
const {value: username, setValue: setUsername} = useStateWithObj
// value를 username으로
// setValue를 setUsername으로 이름 변경
```

ex) 튜플과 배열의 성질 혼합 : 스프렌드 연산자 `...` 를 사용해서, 특정 인덱스에는 요소를 명확한 타입으로 선언하고, 나머지 인덱스에는 배열처럼 개수 제한 없이 받도록 한다.

```
const httpStatusFromPaths: [number, string, ...string[]] = [
  400,
  "Bad Request",
  "/user/:id",
  "users/:userId",
  "users/:uuid",
];
```

옵셔널 ?

특정 속성 or 매개변수가 값이 있을 수도 없을 수도 있다.

```
const optionalTuple1: [number, number, number?] = [1, 2]
```

enum 타입 열거형

일종의 구조체를 만드는 타입 시스템.

- TS는 각 멤버의 값을 스스로 추론하는데, 숫자 0부터 1씩 늘려가며 값을 할당한다.

```
enum ProgrammingLanguage {
  Typescript, // 0
  Javascript, // 1
  Java, // 2
  Python, // 3
  Kotlin, // 4
  Rust, // 5
  Go, // 6
}
```

```
// 각 멤버에게 접근하는 방식 = JS에서 객체의 속성에 접근하는 방식
ProgrammingLanguage.Typescript; // 0
ProgrammingLanguage["Go"]; // 6
```

```
// 역방향 접근 가능
ProgrammingLanguage[2]; // "Java"
```

- 각 멤버에 값을 할당할 수 있고, 일부 멤버에 값을 직접 할당하지 않아도 이전 멤버 값의 숫자를 기준으로 1씩 늘려가며 자동 할당한다.

```
enum ProgrammingLanguage {  
    Typescript = "Typescript",  
    Javascript = "Javascript",  
    Java, // 2  
    Python, // 3  
    Kotlin, // 4  
    Rust, // 5  
    Go, // 6  
}
```

? enum에서 이전 값이 문자열이라면 어떻게 자동할당할까?

🌟 이전 멤버값의 타입에 영향을 받는다!

이전 멤버값이 string일 경우에는 초기화를 해주지 않으면 에러가 난다.

```
enum ProgrammingLanguage {  
    Typescript = "Typescript",  
    Javascript = "Javascript",  
    Java = 3,  
    Python = 2,  
    Kotlin, // 3  
    Rust = "seojin",  
    Go // 초기화 에러  
}
```



enum 타입은 주로 문자열 상수를 생성하는 데 사용된다!

열거형은 관련이 높은 멤버를 모아 **문자열 상수**처럼 사용하고자 할 때 유용하게 쓸 수 있

```
enum ItemStatusType {
    DELIVERY_HOLD = "DELIVERY_HOLD", // 배송 보류
    DELIVERY_READY = "DELIVERY_READY" // 배송 준비 중
    DELIVERING = "DELIVERING" // 배송 중
    DELIVERED = "DELIVERED", // 배송 완료
}

const checkItemAvailable = (itemStatus: ItemStatusType) =>
    switch (itemStatus) {
        // 아래 세 경우는 배송 불가능
        case ItemStatusType.DELIVERY_HOLD:
        case ItemStatusType.DELIVERY_READY:
        case ItemStatusType.DELIVERING:
            return false;

        // 배송 완료되었거나 다른 상태인 경우
        case ItemStatusType.DELIVERED:
        default:
            return true;
    }
}
```

const enum

enum의 역방향 접근 동작을 막기 위한 방법

- 문자열 상수 방식으로 열거형을 사용하는 것이 숫자 상수 방식보다 더 안전하다.

```
const enum NUMBER{
    ONE = 1,
    TWO = 2,
}
```

```
const MyNumber: NUMBER = 100; // NUMBER enum에서 100을 관리하고 있지

const enum STRING_NUMBER {
    ONE = "ONE",
    TWO = "TWO",
}

const myStringNumber: STRING_NUMBER = "THREE"; // 에러
```

?

`const myStringNumber: STRING_NUMBER = "THREE"` 라는 위 구문을 보면, enum에 값 `"THREE"` 이 정의되어 있으면 직접 문자열을 할당할 수 있다는 걸까?

X

STRING_NUMBER enum에 `"ONE"` 이라는 값이 정의되어 있더라도, TypeScript는 직 문자열 `"ONE"` 을 할당하는 것을 허용하지 않는다.

```
const enum STRING_NUMBER {
    ONE = "ONE",
    TWO = "TWO"
}

// 잘못된 사용
const myStringNumber1: STRING_NUMBER = "ONE"; // 에러
const myStringNumber2: STRING_NUMBER = "TWO"; // 에러

// 올바른 사용
const myStringNumber3: STRING_NUMBER = STRING_NUMBER.ONE;
const myStringNumber4: STRING_NUMBER = STRING_NUMBER.TWO;
```

✓ enum의 문제점

- `const enum` 으로 열거형을 선언하더라도 숫자 상수로 관리되는 열거형은 선언한 값 이외의 값을 할당하거나 접근할 때 방지하지 못한다.
- 열거형은 TS 코드가 JS로 변환될 때 즉시 실행 함수 `IIFE` 형식으로 변환된다. 이때 일부 번들러에서 트리셰이킹 과정 중 '즉시 실행 함수로 변환된 값'을 사용하지 않는 코드로 인식하

지 못하는 경우가 발생할 수 있다. → ! 불필요한 코드의 크기가 증가하는 결과 초래

즉시 실행 함수 `IIFE`

: 함수를 정의하자마자 바로 실행하는 JavaScript 패턴

```
// 기본적인 즉시실행함수 형태
(function() {
    // 코드
})();
```



`const enum` 또는 `as const assertion` 사용한다.



`enum` VS. `const enum` VS. `as const`

`enum`

- JavaScript로 컴파일될 때 즉시실행함수로 변환
- 실제 객체가 생성됨
- 런타임에서 존재
- 번들 사이즈가 커질 수 있음

`const enum`

- 컴파일 시점에 실제 값으로 대체됨
- 런타임에 객체가 생성되지 않음
- **번들 사이즈 최적화**
- 객체로서의 기능 사용 불가 (`Object.keys` 등)

`as const`

: **리터럴 타입**을 사용하여 값을 변경 불가능하도록 만드는 방법

리터럴 타입 : 특정 값에 대한 타입을 명시. 즉, 변수나 값이 가질 수 있는 구체적인 값을 타입으로 지정하는 것.

- 일반 객체로 컴파일됨
- 모든 속성이 `readonly`
- 객체로서의 모든 기능 사용 가능
- 타입 추론이 더 정확함

```
// 1. enum
enum Direction1 { Up = "UP", Down = "DOWN" }
const d1 = Direction1.Up; // "UP"

// 2. const enum
const enum Direction2 { Up = "UP", Down = "DOWN" }
const d2 = Direction2.Up; // 컴파일 시 "UP"으로 직접 대체

// 3. as const
const Direction3 = {
  Up: "UP",
  Down: "DOWN"
}
```



```


    } as const;
    const d3 = Direction3.Up;  // "UP"

    const direction = ['Up', 'Down', 'Left'] as const;
    // direction의 타입은 ['Up', 'Down', 'Left']로 고정된다.

```

타입 조합

교차 타입

 여러 가지 타입을 결합하여 하나의 단일 타입으로 만들 수 있다.

```

type ProductItem = {
  id: number;
  name: string;
  type: string;
  price: number;
  imageUrl: string;
  quantity: number;
};

type ProductItemWithDiscount = ProductItem & { discountAmount: number };

```



interface에서의 교차 타입

- type은 `&` 연산자 사용
- interface는 `extends` 키워드 사용

▼ 예시

```
// type으로 교차 타입 사용
type Animal = {
  name: string;
}

type Bear = Animal & {
  honey: boolean;
}

// interface로 교차 타입 사용
interface Car {
  brand: string;
}

interface ElectricCar extends Car {
  battery: number;
}

// interface와 type 혼합 사용
interface Dog {
  bark: () => void;
}

type Pet = Animal & Dog;

// 실제 사용 예시
const winnieThePooh: Bear = {
  name: "Winnie",
  honey: true
};


const tesla: ElectricCar = {
  brand: "Tesla",
```

```

        battery: 100
    };

    const myDog: Pet = {
        name: "Max",
        bark: () => console.log("Woof!")
    };

```

 타입을 더할 때는 교차타입을, 특정 속성을 뺄 때는 Omit을 사용한다.


유니온 타입


교차 타입은 `A & B` 모두 만족하는 경우라면, 유니온 타입은 `A | B` 타입A 또는 타입B 중 하나가 될 수 있는 타입을 말한다.

```

type CardItem = {
    id: number;
    name: string;
    type: string;
    imageUrl: string;
};

type PromotionEventItem = ProductItem | CardItem;

const printPromotionItem = (item: PromotionEventItem) => {
    console.log(item.name) // 

    console.log(item.quantity); //  컴파일 에러 발생
    // quantity가 ProductItem에만 존재하기 때문이다.
    // PromotionEventItem은 CardItem도 포함하는데, CardItem은 quant
    // PromotionEventItem은 quantity를 참조할 수 X다.
}

```

? 그러면 어떻게 해결해야할까?
타입가드를 사용한다.

타입 가드 : TS에서 조건문을 사용해 타입의 범위를 좁히는 방법
➤ typeof 타입 가드, instanceof 타입 가드, in 타입 가드, 사용자의 타입 가드

is

- in 연산자 사용

```
const printPromotionItem = (item: PromotionEventItem) => {
  console.log(item.name); // 공통 속성은 바로 사용 가능

  if ('quantity' in item) {
    // 여기서는 item이 ProductItem으로 타입이 좁혀짐
    console.log(item.quantity);
  }
}
```

인덱스 시그니처

특정 타입의 속성 이름은 알 수 없지만, 속성값의 타입을 알고 있을 때 사용한다.

인터페이스 내부에 `[Key: K]: T` 꼴로 타입을 명시해주면 된다.

- 해당 타입의 속성 키는 모두 K 타입이어야 하고, 속성값은 모두 T 타입을 가져야 한다는 의미이다.

```
interface IndexSignatureEx {
  [key: string]: number;
}
```

다른 속성을 추가로 명시해줄 수 있는데 이때 추가로 명시된 속성은 인덱스 시그니처에 포함되는 타입이어야 한다.

```
interface IndexSignatureEx2 {
  [key: string]: number | boolean;
}
```

```
length: number;
isValid: boolean;
name: string; // 에러 발생
}
```

▼ 추가 예시 (활용)

```
interface StringNumberMap {
    [key: string]: number;
}

const scores: StringNumberMap = {
    math: 90,
    science: 85,
    history: 95
    // english: "A" // 에러: 모든 값은 숫자여야 함
};
```

인덱스드 액세스 타입

다른 타입의 특정 속성이 가지는 타입을 조회하기 위해 사용된다.

```
type Example = {
    a: number;
    b: string;
    c: boolean;
}

type IndexedAccess = Example["a"]; // "a" 속성의 타입만 가져온다 -
type IndexedAccess2 = Example["a" | "b"]; // "a"와 "b" 속성의 타입
type IndexedAccess3 = Example[keyof Example] // number | string
// keyof Example은 "a" | "b" | "c"를 의미

type ExAlias = "b" | "c";
type IndexedAccess4 = Example[ExAlias]; // string | boolean
// "b"와 "c" 속성의 타입을 유니온으로 가져온다
```

- 배열의 요소 타입을 조회하기 위해 사용한다.

- 배열의 인덱스는 숫자 타입이므로 number로 인덱싱하여 배열 요소를 얻은 다음에 typeof 연산자를 붙여주면, 해당 배열 요소의 타입을 가져올 수 있다.

```
const PromotionList = [
  {type: "product", name: "chicken"},
  {type: "product", name: "pizza"},
  {type: "card", name: "cheer-up"},
];

type ElementOf<T> = typeof T[number];
// typeof T: 값의 타입을 가져옴
// [number]: 배열의 요소 타입을 가져옴. 즉, 배열에서 하나의 요소를 꺼냈을 때

// type PromotionItemType = {type: string; name: string} (결과)
type PromotionItemType = ElementOf<PromotionList>;
```

실제 동작 과정

```
// 1단계: typeof PromotionList
type Step1 = typeof PromotionList;
// {type: string; name: string}[]

// 2단계: [...][number]
type Step2 = Step1[number];
// {type: string; name: string}
```

공식문서

```
const MyArray = [
  { name: "Alice", age: 15 },
  { name: "Bob", age: 23 },
  { name: "Eve", age: 38 },
];

type Person = typeof MyArray[number];
// type Person = {
//   name: string;
//   age: number;
// }
type Age = typeof MyArray[number]["age"];
```

```
// type Age = number

// Or
type Age2 = Person["age"];
// type Age2 = number
```

맵드 타입

JS의 `map` 메서드 : 배열 A를 기반으로 새로운 배열 B를 만들어내는 배열 메서드

맵드 타입 : 다른 타입을 기반으로 한 타입을 선언할 때 사용하는 문법

- **인덱스 시그니처** 문법을 사용해서 반복적인 타입 선언을 효과적으로 줄일 수 있다.

```
type Example = {
  a: number;
  b: string;
  c: boolean;
};

type Subset<T> = {
  [K in keyof T]?: T[K]; // 모든 속성을 optional로 만들
  // keyof T: T의 모든 키를 유니온 타입으로 가져옴
  // keyof Example은 "a" | "b" | "c"

  // K in : 위에서 얻은 각 속성 이름을 순회
  // K는 차례대로 "a", "b", "c"가 됨

  // T[K]: 각 키에 해당하는 각 속성의 타입을 가져옴
}

const aExample: Subset<Example> = {a: 3};
const bExample: Subset<Example> = {b: "hello"};
const acExample: Subset<Example> = {a: 4, c: true};
```

✓ 맵드 타입에서 매핑할 때 적용하는 수식어

- **readonly** : 읽기 전용으로 만들고 싶을 때 붙여주는 수식어
- **?** : 선택적 매개변수로 만들고 싶을 때 붙여주는 수식어

- `-` : readonly나 ? 앞에 -를 붙여주면 해당 수식어를 제거한 타입을 선언할 수 있다.

```
type ReadOnlyEx = {
    readonly a: number;
    readonly b: string;
};

type CreateMutable<Type> = {
    -readonly [Property in keyof Type]: Type[Property];
    // 이때 배열 예시랑 다르게 [number]를 하지 않는 이유는 배열의 인덱스가
    // a: number, b: string
};

type ResultType = CreateMutable<ReadOnlyEx>; // { a: number, b: string }

type OptionalEx = {
    a?: number;
    b?: string;
    c: boolean;
};

type Concrete<Type> = {
    [Property in keyof Type]-?: Type[Property];
};

type ResultType = Concrete<OptionalEx>; // { a: number; b: string; c: boolean }
```

맵드 타입이 실제로 사용된 예시 : 배달의민족 선물하기 서비스

배달의민족 선물하기 서비스에는 `바텀시트` 라는 컴포넌트가 존재한다. 밑에서부터 스크롤 올라오는 모달이다.

- 이 바텀시트는 선물하기 서비스의 최근 연락처 목록, 카드 선택, 상품 선택 등 여러 지면에서 사용되고 있다.
- 바텀시트마다 각각 `resolver`, `isOpened` 등의 상태를 관리하는 스토어가 필요한데, 이 스토어의 타입 `BottomSheetStore` 을 선언해줘야 한다.

! 이때 `BottomSheetMap` 에 존재하는 모든 키에 대해 일일이 스토어를 만들어줄 수도 있지만 불필요한 반복 발생

👤 인덱스 시그니처 문법을 사용해서 BottomSheetMap을 기반으로 각 키에 해당하는 스토어를 선언할 수 있다.

```
// BottomSheetMap은 바텀시트의 종류와 해당하는 컴포넌트를 매핑한 객체이다.
const BottomSheetMap = {
  RECENT_CONTACTS: RecentContactsBottomSheet, // 최근 연락처
  CARD_SELECT: CardSelectBottomSheet, // 카드 선택
  SORT_FILTER: SortFilterBottomSheet, // 정렬/필터
  PRODUCT_SELECT: ProductSelectBottomSheet, // 상품 선택
  REPLY_CARD_SELECT: ReplyCardSelectBottomSheet, // 답장 카드
  RESEND: ResendBottomSheet, // 재전송 바텀시트
  STICKER: StickerBottomSheet, // 스티커 바텀시트
  BASE: null, // 기본 바텀시트
};

export type BOTTOM_SHEET_ID = keyof typeof BottomSheetMap; // "RECENT_CONTACTS"

// ! 불필요한 반복 발생
type BottomSheetStore = {
  RECENT_CONTACTS: {
    resolver?: (payload: any) => void;
    args?: any;
    isOpened: boolean;
  };
  CARD_SELECT: {
    resolver?: (payload: any) => void;
    args?: any;
    isOpened: boolean;
  };
  SORT_FILTER: {
    resolver?: (payload: any) => void;
    args?: any;
    isOpened: boolean;
  };
  // ...
};

// ✅ Mapped Types를 통해 효율적으로 타입을 선언할 수 있다.
type BottomSheetStore = {
  [index in BOTTOM_SHEET_ID]: {
```

```

    resolver?: (payload: any) => void;
    args?: any;
    isOpened: boolean;
  };
};

```



React에서 컴포넌트를 객체로 매핑하는 패턴

```

// 각각의 바텀시트 컴포넌트들
const RecentContactsBottomSheet = () => {
  return <div>최근 연락처</div>
}

const CardSelectBottomSheet = () => {
  return <div>카드 선택</div>
}

// 컴포넌트 매핑
const BottomSheetMap = {
  RECENT_CONTACTS: RecentContactsBottomSheet,
  CARD_SELECT: CardSelectBottomSheet,
};

// 사용 예시
function App() {
  const sheetId = 'RECENT_CONTACTS';
  const SelectedSheet = BottomSheetMap[sheetId];

  return <SelectedSheet />;
}

```

✓ 맵드 타입에서 as 키워드 사용

맵드 타입에서는 as 키워드를 사용하여 키를 재지정할 수 있다.

BottomSheetStore의 키 이름에 BottomSheetMap의 키 이름을 그대로 쓰지 않고, 모든 키에 `BOTTOM_SHEET` 을 붙이는 식으로 공통된 처리를 적용해 새로운 키를 지정할 수 있다.

```
type BottomSheetStore = {
  [index in BOTTOM_SHEET_ID as `${index}_BOTTOM_SHEET`]: {
    resolver?: (payload: any) => void;
    args?: any;
    isOpened: boolean;
  };
};
```

템플릿 리터럴 타입

JS의 템플릿 리터럴 문자열을 사용하여 문자열 리터럴 타입을 선언할 수 있는 문법

ex) (위의 예시)BottomSheetMap의 각 키에다 `_BOTTOM_SHEET`을 붙여주는 예시

```
type Stage =
  | "init"
  | "select-image"
  | "edit-image"
  | "decorate-card"
  | "captrue-image";

type StageName = `${Stage}-stage`; // 새로운 문자열 리터럴 유니온 타입
// 'init-stage' | 'select-image-stage' | 'edit-image-stage' | 'c
```

제네릭

일반화된 데이터 타입. `<T>`

| `T` `Type` , `E` `Element` , `K` `Key` , `V` `Value`

함수, 타입, 클래스 등에서 내부적으로 사용할 타입을 미리 정해두지 않고, 타입 변수를 사용해서 해당 위치를 비워 둔 다음에, 실제로 그 값을 사용할 때 외부에서 타입 변수 자리에 타입을 지정하여 사용하는 방식

- 정적 언어 `C`, `Java`, `TS` 에서 다양한 타입 간에 재사용성을 높이기 위해 사용하는 문법

```
type StringArrayType = string[]; // 문자열 배열을 위한 타입

const stringArray: StringArrayType = ["치킨", "피자", "우동"];
```

```
type ExampleArrayType<T> = T[];

const array1: ExampleArrayType<String> = ["치킨", "피자", "우동"];
```

- 제네릭 함수를 호출할 때 반드시 `<>` 안에 타입을 명시해야 하는 것은 아니다. 타입 명시 부분을 생략하면 컴파일러가 인수를 보고 타입을 추론해준다. 즉, 타입 명시 생략이 가능하다.

```
function exampleFunc<T>(arg: T): T[] {
    return new Array(3).fill(arg);
}


exampleFunc("hello"); // T는 string으로 추론한다.
// exampleFunc<string>("hello");
```

- 제네릭 타입에 기본값을 추가할 수 있다.

```
interface SubmitEvent<T = HTMLElement> extends SyntheticEvent<T>
```

HTMLElement : 웹 페이지의 모든 HTML 요소의 기본 인터페이스
SyntheticEvent : React에서 제공하는 이벤트 래퍼(wrapper)

! 제네릭을 사용할 때 어떤 타입이든 될 수 있기 때문에 배열에만 존재하는 `length` 속성을 제네릭에서 참조하려고 하면 당연히 에러가 발생한다.

 제네릭 `<>` 내부에 `length` 속성을 가진 타입만 받는다 라는 제약을 걸어줘서 `length` 속성을 사용할 수 있게 만든다.


```
interface TypeWithLength {
    length: number;
}

function exampleFunc2<T extends TypeWithLength>(arg: T): number
```

```
    return arg.length;
}
```

! 파일 확장자가 `tsx` 일 때 화살표 함수에 제네릭을 사용하면 에러가 발생한다.

tsx는 타입스크립트 + JSX이기 때문에 제네릭의 `<>`와 태그의 `<>`를 혼동하여 문제가 발생하는 것이다.

 제네릭 부분에 `extends` 키워드를 사용하여 컴파일러에게 특정 타입의 하위 타입만 올 수 있음을 확실히 알려주면 된다.

- 보통 제네릭을 사용할 때는 `function` 키워드로 선언하는 경우가 많다.

```
// 에러 발생: JSX element 'T' has no corresponding closing tag
const arrowExampleFunc = <T>(arg: T): T[] => {
    return new Array(3).fill(arg);
};

// 에러 발생 x
const arrowExampleFunc2 = <T extend {}>(arg: T): T[] => {
    return new Array(3).fill(arg);
};
```