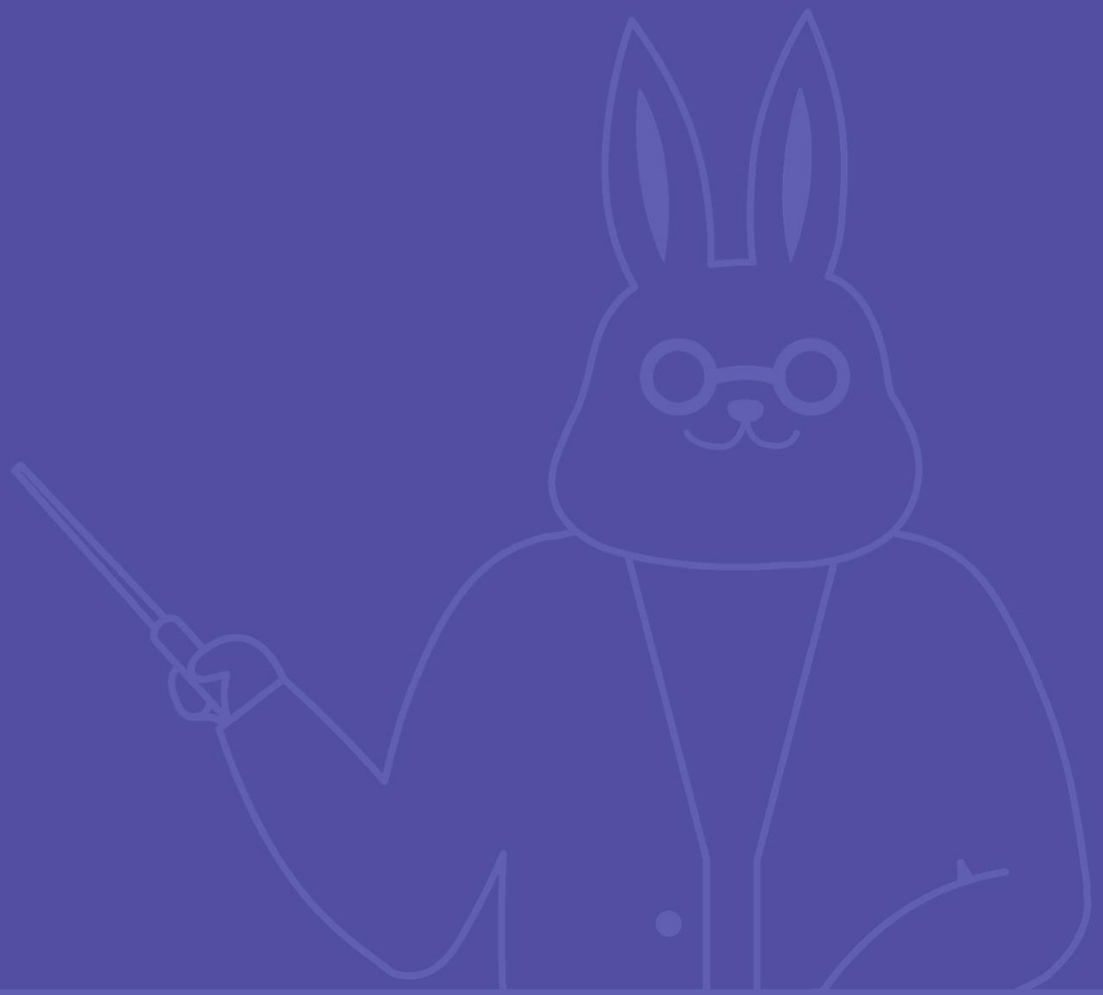
타입스크립트 II

01 타입 심화



/* elice */

커리큘럼

타입 심화

다양한 예제와 실무에서 쓰는 코드를 통해 타입을 좁히는 방법과 확장하는 방법을 알아보고, 타입스크립트를 더 자유자재로 쓸 수 있게 됩니다.

Decorator

Decorator가 필요한 이유와 작성 방법, 특징을 알고 Decorator 타입에 따라 알맞은 용도에 쓸 수 있게 됩니다.

커리큘럼

NodeJS와 Typescript

자바스크립트로 작성된 nodejs, express 서버 코드를 타입스크립트로 마이그레이션 합니다. 오픈소스 코드를 뜯어보며 타입스크립트의 데코레이터와 인터페이스를 중점으로 사용하는 nestjs에 대한 이론 및 실습을 진행합니다.

React와 Typescript

자바스크립트로 작성된 react 코드를 타입스크립트로 마이그레이션 해보면서 달라지는 점을 확인합니다. 간단한 프로젝트를 개발하면서 리액트에서 타입스크립트가 어떻게 사용될 수 있는지 실습합니다.

💡 추천대상

1. 자바스크립트를 사용하는 실무자

프로젝트에서 자바스크립트를 사용하는데 타입스크립트로 마이그레이션 해야 하는 필요성을 느껴서 실무에서 타입스크립트를 잘 적용하는 방법을 배우고 싶으신 분들께 추천드립니다.

2. 타입스크립트 기초를 배운 초보자

이 강의는 다양하면서도 쉬운 예제로 구성된 실습들로 이루어져 있습니다. 타입, 인터페이스 등을 선언할 수 있지만 오버로딩이나 데코레이터 등 심화 기능에 대해 이해하기 어렵다고 느껴지시는 분들께 추천해 드립니다.

3. 프론트엔드, 백엔드 신입 개발자

이 강의는 타입스크립트를 사용하는 곳에 지원하신 분들이 과제를 해결하거나 또는 취업하신 분들이 실무에서 사용되는 타입스크립트 코드를 이해하는 데 도움이 될 것입니다. 더 나아가 리액트와 nestjs 각 프로젝트에 타입스크립트 기능을 적재적소에 쓰는 방법을 알고 싶으신 분께 추천해 드립니다.

🎯 수강목표

1. 타입 좁히기

타입 가드를 통해 방어적인 코드를 작성하는 방법을 익힙니다.

2. 타입 확장하기

오버로딩, 유니온 등을 통해 유연한 타이핑을 하는 방법을 배웁니다.

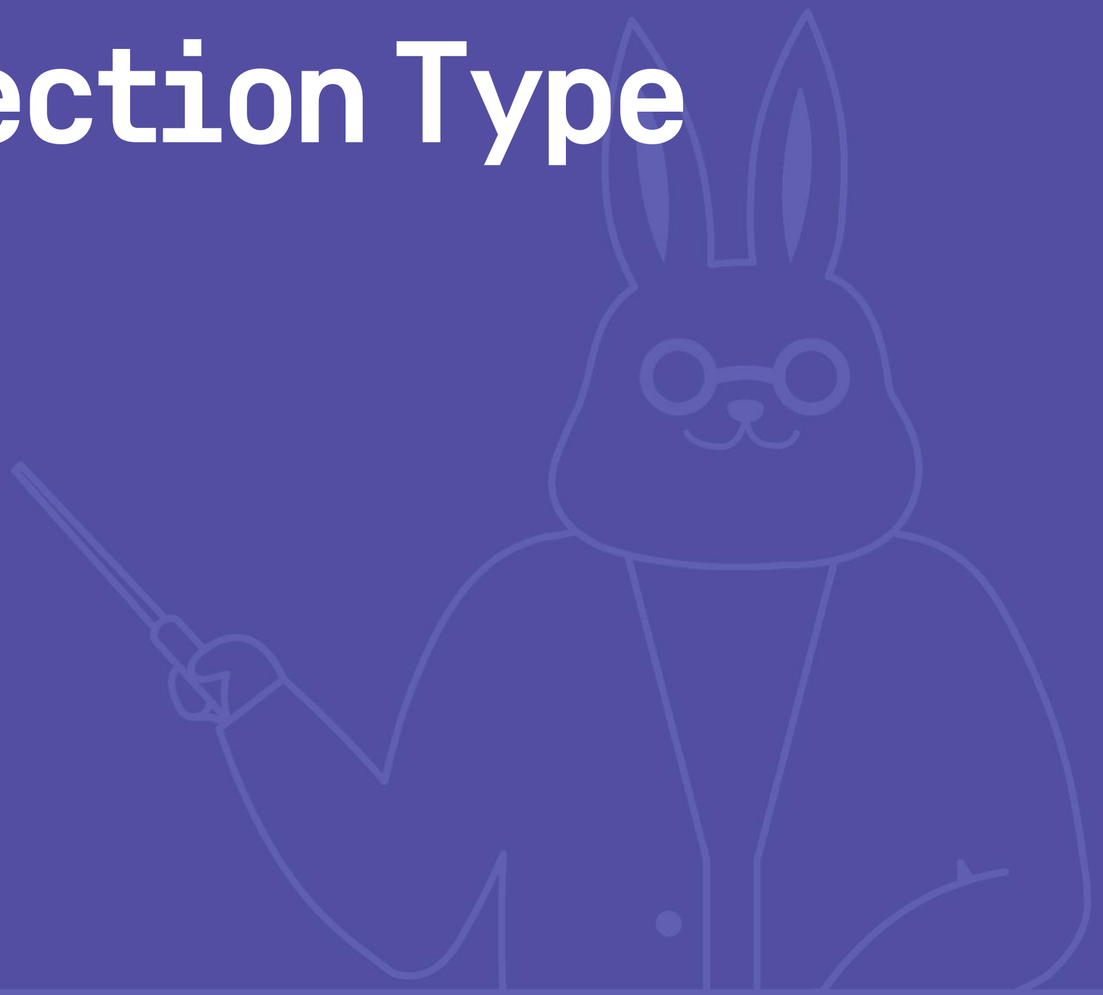
3. 타입 기능 익히기

타입의 기능에 대해 다양한 주의사항을 숙지합니다.

목차

01

Union Type, Intersection Type



코드

```
interface Animal {
  eat: () => void
  sleep: () => void
}

class Dog implements Animal {
  eat() {}
  sleep() {}
}

class Cat implements Animal {
  eat() {}
  sleep() {}
}
```

Dog에만 **bark** 라는 메서드를 추가할 때,
Animal에 **bark** 메서드를 추가하면
interface를 구현하는 클래스는
구현의 의무가 있기 때문에
Cat까지 bark 메서드를 구현해야 함

기존 타입, 인터페이스의 변경은
이미 그 타입을 사용하고 있는 코드에
똑같은 변경을 가해줘야 함

만약 해당 타입을 쓰는 모든 코드에 변경을 가하지 않고
특정 코드만 **자유롭게 타입을 확장하고 싶을 땐** 어떻게 해야 할까?

Intersection ——→ And **A 타입이면서 B 타입**

Union ——→ Or **A 타입, B 타입 둘 중 하나**

Union Type

코드

```
let one: string | number;

one = '1';
one = 1;
```

- '|' 바 사용
- Type A, Type B가 있을 때 A와 B를 유니온하면, A 타입 또는 B 타입 둘 중 하나
- one: string 이거나 number 둘 중 하나

Union Type 언제 사용할까?

코드

```ts
type A = string | number;

// An interface can only extend an object type
// or intersection of object types with statically known members
interface StrOrNum extends A {
  a: string;
}

// Ok
type StrOrNum = {
  a: string;
} & (string | number);
```

- 여러 타입 중 하나가 올 것이라고 가정할 때 사용

- 단, 인터페이스에 유니온 타입을 사용하는 경우 인터페이스는 유니온 타입을 확장하지 못함
- 이럴 때는 type과 &를 사용해야 함

Union Type 주의할 점: 동시에 여러 타입이 될 수 없다

코드

```
type Human = {
  think: () => void;
};

type Dog = {
  bark: () => void;
};
declare function getEliceType(): Human |
Dog;

const elice = getEliceType();
elice.think(); // Property 'think' does
not exist on type 'Dog'.
elice.bark(); // Property 'bark' does not
exist on type 'Human'.
```

- elice를 Human 또는 Dog 라는 유니온 타입으로 만들어 주어도 think, bark 둘 중 어느 것도 사용하지 못함

- 왜? **Human인지 Dog인지 확신할 수 없기 때문**
- elice.think() 함수를 호출할 때 elice가 Human이라면 괜찮지만 Dog라면 think를 호출할 수 없기 때문에 컴파일 단계에서 에러

- 해결 방법: **타입 가드**

Intersection Type

코드

```
type Human = {
  think: () => void;
};
type Developer = {
  work: () => void;
};

const elice: Human & Developer = {
  think() {
    console.log(`I'm thinking`);
  },
  work() {
    console.log(`I'm working`);
  },
};
```

- 교차 타입, Intersection Type
- &(앰퍼샌드) 사용
- Type A, Type B가 있을 때 A와 B를 인터섹션 하면, A 타입이면서 B 타입이라는 의미
- Elice: Human이면서 Developer

Intersection Type 언제 사용할까?

코드

```
type Human = {
  think: () => void;
};
type Developer = {
  work: () => void;
};
type Student = {
  study: () => void;
};

const elice: Human & Developer & Student =
{
  think() {
    console.log(`I'm thinking`);
  },
  work() {
    console.log(`I'm working`);
  },
  study() {
    console.log(`I'm studying`);
  },
};
```

- Intersection Type은 **기존 타입을 대체하지 않으면서 기존 타입에 새로운 필드를 추가하고 싶을 때** 사용
- 기존의 Human & Developer 타입의 elice에 Student 타입을 추가해 기존 타입은 변경하지 않고 확장 가능

Intersection Type 주의할 점: 각 타입에 겹치는 필드가 있다면 어떻게 될까?

타입이 같을 때

```ts
type Developer = {
  output: string;
  develop: () => void;
};
type Designer = {
  output: string;
  design: () => void;
};

const 개자이너: Developer & Designer
= {
  output: 'something cool',
  develop() {
    console.log(`I'm working`);
  },
  design() {
    console.log(`I'm working`);
  },
};
```

타입이 다를 때

```ts
type Developer = {
  output: number;
  develop: () => void;
};
type Designer = {
  output: string;
  design: () => void;
};

const 개자이너: Developer & Designer
= {
  output: 'something cool',
  develop() {
    console.log(`I'm working`);
  },
  design() {
    console.log(`I'm working`);
  },
};
```

타입이 포함관계일 때

```ts
type Developer = {
  output: number;
  develop: () => void;
};
type Designer = {
  output: string | number;
  design: () => void;
};

const 개자이너: Developer & Designer
= {
  // output: 'something cool',
  output: 1,
  develop() {
    console.log(`I'm working`);
  },
  design() {
    console.log(`I'm working`);
  },
};
```

Intersection Type 주의할 점: 각 타입에 겹치는 필드가 있다면 어떻게 될까?

타입이 같을 때 : Ok

```ts
type Developer = {
  output: string;
  develop: () => void;
};
type Designer = {
  output: string;
  design: () => void;
};

const 개자이너: Developer & Designer
= {
  output: 'something cool',
  develop() {
    console.log(`I'm working`);
  },
  design() {
    console.log(`I'm working`);
  },
};
```

타입이 다를 때 : Error

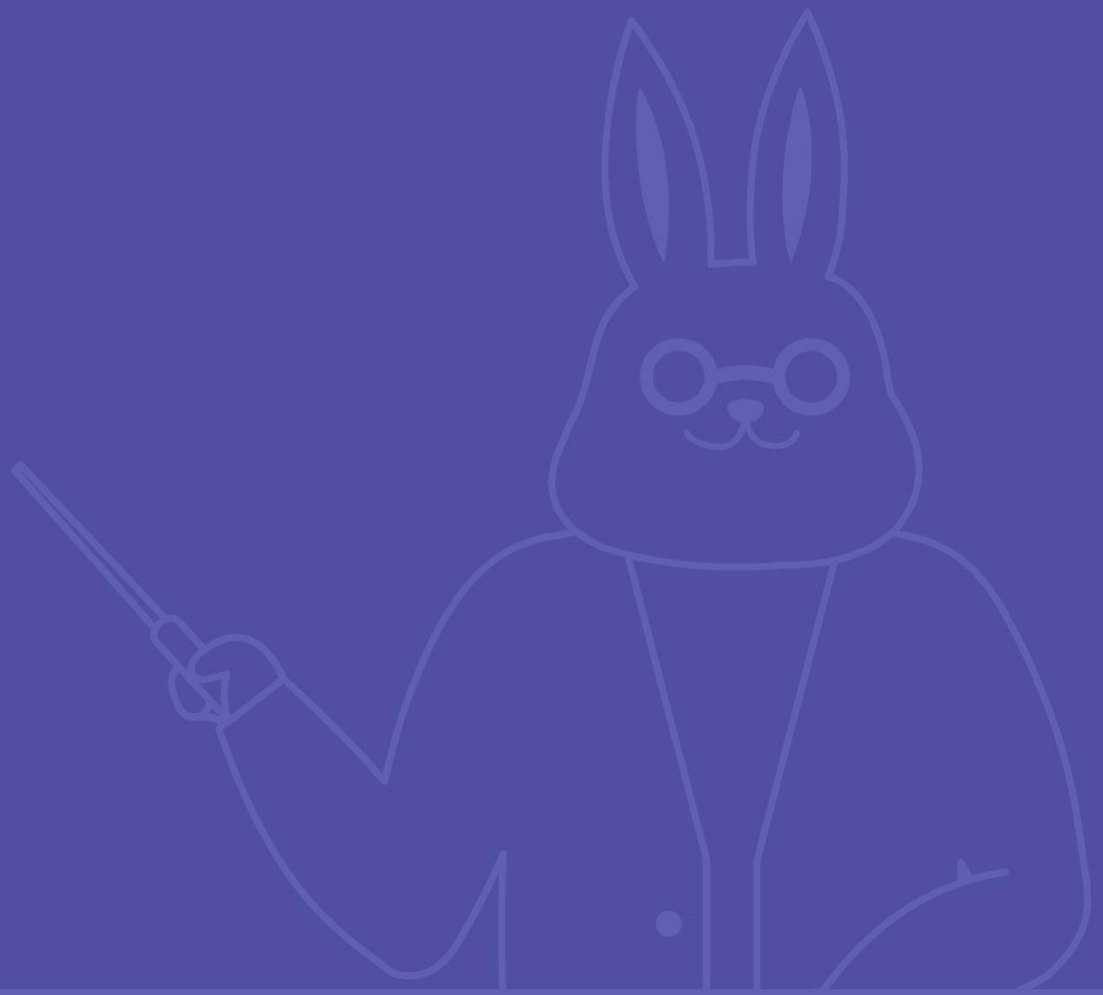
```ts
type Developer = {
  output: number;
  develop: () => void;
};
type Designer = {
  output: string;
  design: () => void;
};

const 개자이너: Developer & Designer
= {
  output: 'something cool',
  develop() {
    console.log(`I'm working`);
  },
  design() {
    console.log(`I'm working`);
  },
};
```

타입이 포함관계일 때: Ok or Error

```ts
type Developer = {
  output: number;
  develop: () => void;
};
type Designer = {
  output: string | number;
  design: () => void;
};

const 개자이너: Developer & Designer
= {
  // output: 'something cool',
  // error
  output: 1, // ok
  develop() {
    console.log(`I'm working`);
  },
  design() {
    console.log(`I'm working`);
  },
};
```

02

Type Guard



Type Guard

데이터의 타입을 알 수 없거나, 될 수 있는 타입이 여러 개라고 가정할 때
조건문을 통해 데이터의 타입을 좁혀나가는 것.

데이터의 **타입에 따라 대응하여 에러를 최소화**할 수 있음.
타입을 통해 '가드'하는 코드, 방어적인 코드를 짤 수 있음

타입 가드를 사용해 유니온 타입을 제대로 사용해보자.

구별된 유니온(Discriminated Union)

```
type Human = {
  think: () => void;
};

type Dog = {
  bark: () => void;
};

declare function getEliceType(): Human |
Dog;

const elice = getEliceType();
```

- elice가 Human인지, Dog인지 확신할 수 없는 상태.
- 타입스크립트가 **타입을 추론할 수 있도록 단서를 줘보면 어떨까?**

구별된 유니온

- 타입을 구별할 수 있는 단서가 있는 유니온 타입
- 구별된 유니온, 태그된 유니온, 서로소 유니온

타입 가드를 사용해 유니온 타입을 제대로 사용해보자.

구별된 유니온(Discriminated Union)

```ts
type Human = {
  think: () => void;
};

type Dog = {
  tail: string;
  bark: () => void;
};

declare function getEliceType(): Human | Dog;

const elice = getEliceType();

if ('tail' in elice) {
  elice.bark();
} else {
  elice.think();
}
```

구별된 유니온으로 타입 가드 하는 방법

1. 각 타입에 타입을 구별할 단서(태그)를 만든다.
2. 조건문을 통해 각 타입의 단서로 어떤 타입인지 추론한다.
3. 해당 타입의 프로퍼티를 사용한다.

타입 가드를 사용해 유니온 타입을 제대로 사용해보자.

구별된 유니온(Discriminated Union)

```
type SuccessResponse = {
  status: true;
  data: any;
};
type FailureResponse = {
  status: false;
  error: Error;
};

type CustomResponse = SuccessResponse |
FailureResponse;

declare function getData(): CustomResponse;

const response: CustomResponse = getData();

if (response.status) {
  console.log(response.data);
} else if (response.status === false) {
  console.log(response.error);
}
```

실무에서 자주 쓰는 구별된 유니온과 타입 가드

- 서버에서 오는 응답 또는 함수의 결과가 여러 갈래로 나뉠 때 구별된 유니온 사용 가능
- 타입의 단서를 토대로 타입 가드를 하고, 응답의 결과에 따라 다른 작업을 실행시켜준다

다양한 연산자를 통한 타입 가드

01 instanceof

```
class Developer {
  develop() {
    console.log(`I'm working`);
  }
}
class Designer {
  design() {
    console.log(`I'm working`);
  }
}

const work = (worker: Developer | Designer) => {
  if (worker instanceof Designer) {
    worker.design();
  } else if (worker instanceof Developer) {
    worker.develop();
  }
};
```

instanceof

- 클래스도 타입. 객체가 어떤 클래스의 객체인지 구별할 때 사용하는 연산자
- '인스턴스 instanceof 클래스'와 같이 사용

다양한 연산자를 통한 타입 가드

02 typeof

```
const add = (arg?: number) => {
  if (typeof arg === 'undefined') {
    return 0;
  }

  return arg + arg;
};
```

typeof

-데이터의 타입을 반환하는 연산자
-typeof 데이터 === ‘string’ 과 같이 사용
-typeof 데이터 === ‘undefined’ 처럼 undefined 체크도 가능
-참고로 ‘데이터 == null’과 같이 쓰면 null, undefined 둘 다 체크
-올해 고쳐야 할 타입스크립트 10가지 나쁜 습관에 의하면 null과 undefined는 따로 체크해주는 게 더 명확함

다양한 연산자를 통한 타입 가드

03 in

```ts
type Human = {
  think: () => void;
};

type Dog = {
  tail: string;
  bark: () => void;
};

declare function getEliceType(): Human | Dog;

const elice = getEliceType();

if ('tail' in elice) {
  elice.bark();
} else {
  elice.think();
}
```

in

- 문자열 A in 오브젝트: 오브젝트의 key 중에 문자열 A가 존재하는가

다양한 연산자를 통한 타입 가드

04 literal type guard

```
type Action = 'click' | 'hover' | 'scroll';

const doPhotoAction = (action: Action) => {
  switch (action) {
    case 'click':
      showPhoto()
      break;
    case 'hover':
      zoomInPhoto()
      break;
    case 'scroll':
      loadPhotoMore()
      break;
  }
}

// or

const doPhotoAction2 = (action: Action) => {
  if (action === 'click') {
    showPhoto();
  } else if (action === 'hover') {
    zoomInPhoto();
  } else if (action === 'scroll') {
    loadPhotoMore();
  }
};
```

literal type guard

- 리터럴 타입: 특정 타입의 하위 타입. 구체적인 타입.
- Ex) string 타입의 리터럴 타입: ‘cat’, ‘dog’, …
- 리터럴 타입은 **동등(==), 일치(===) 연산자** 또는 **switch**로 타입 가드 가능

참고로 예제처럼 하나의 value 값에 따라 조건문을 작성하는 경우, 대부분의 경우 switch의 연산 결과가 if-else보다 더 빠르므로 되도록 switch를 쓰는 걸 권장해 드립니다. 조건문의 개수가 적으면 큰 차이는 없지만, 조건문의 개수가 많아질수록 switch의 성능이 더 빠릅니다.

https://www.oreilly.com/library/view/high-performance-javascript/9781449382308/ch04.html#if-else_versus_switch

다양한 연산자를 통한 타입 가드

05 사용자 정의 함수

```
import is from '@sindresorhus/is';

const getString = (arg?: string) => {
  if (is.nullOrUndefined(arg)) {
    return '';
  }

  if (is.emptyStringOrWhitespace(arg)) {
    return '';
  }

  return arg;
};
```

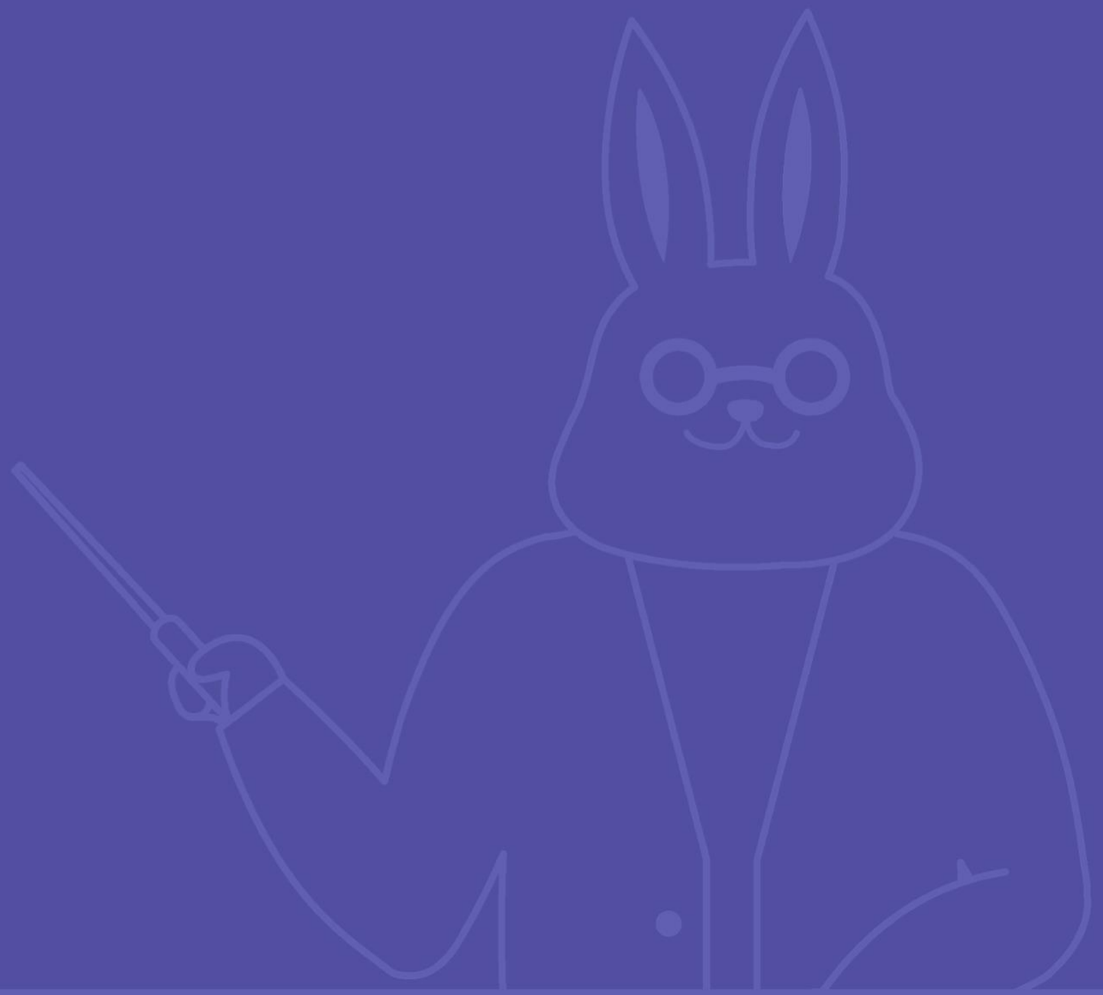
Type guard에 유용한 오픈소스

- 사용자 정의 함수를 사용해 타입 가드 가능
- 오픈소스 중 sindresorhus/is를 사용하여 가독성 있게 타입 체크 가능
- Yarn add @sindersorhus/is 또는 npm install @sindersorhus/is

- https://github.com/sindresorhus/is

03

Optional Chaining



Optional Chaining

접근하는 객체의 프로퍼티가 **null 또는 undefined일 수 있는**
optional property인 경우 if 문을 사용하지 않고 넘어가게 하는 체이닝 방법

es2020에서 추가된 문법이며 **객체가 null 또는 undefined이면**
undefined를 리턴. 그렇지 않은 경우 데이터 값을 리턴

&&와 ?.의 차이점

falsy(false, null, undefined, ‘’, 0, -0, NaN) 값 체크

&&

```
type Dog = {
  hasTail: boolean;
  살랑살랑: () => string;
};

/**
 * hasTail이 true인 경우 반환값은 string이지만
 * false인 경우 && 연산자(and 연산자)에 의해 dog.hasTail을 반환
 * 따라서 의도와는 달리 string 또는 false를 반환하게 됨
 */
function 꼬리살랑살랑(dog: Dog): string {
  // -> string | false
  return dog.hasTail && dog.살랑살랑();

 // 이렇게 써야 함
 // if (dog.hasTail) {
 //   dog.살랑살랑();
 // }
}
```

?.는 null과 undefined만 체크.

Optional Chaining

```
type Dog = {
  tail?: {
    살랑살랑: () => string;
  };
};

function 꼬리살랑살랑(dog: Dog): string {
  return dog.tail?.살랑살랑();
}
```

Optional Chaining 활용

Object, Array

```ts
/**
 * Object
 */
type CustomResponse = {
  data: any;
};

const findData = (response?: CustomResponse) => {
  return response?.data;
};

/**
 * Array
 */
type Post = {
  comments?: string[];
};

const getFirstComment = (post: Post) => {
  return post.comments?.[0];
};
```

Optional Function

```ts
type Cat = {
  sitDown?: () => void;
};

function trainCat(cat: Cat) {
  cat.sitDown?.();
}
```

Optional Chaining으로 if문을 줄여보자

If

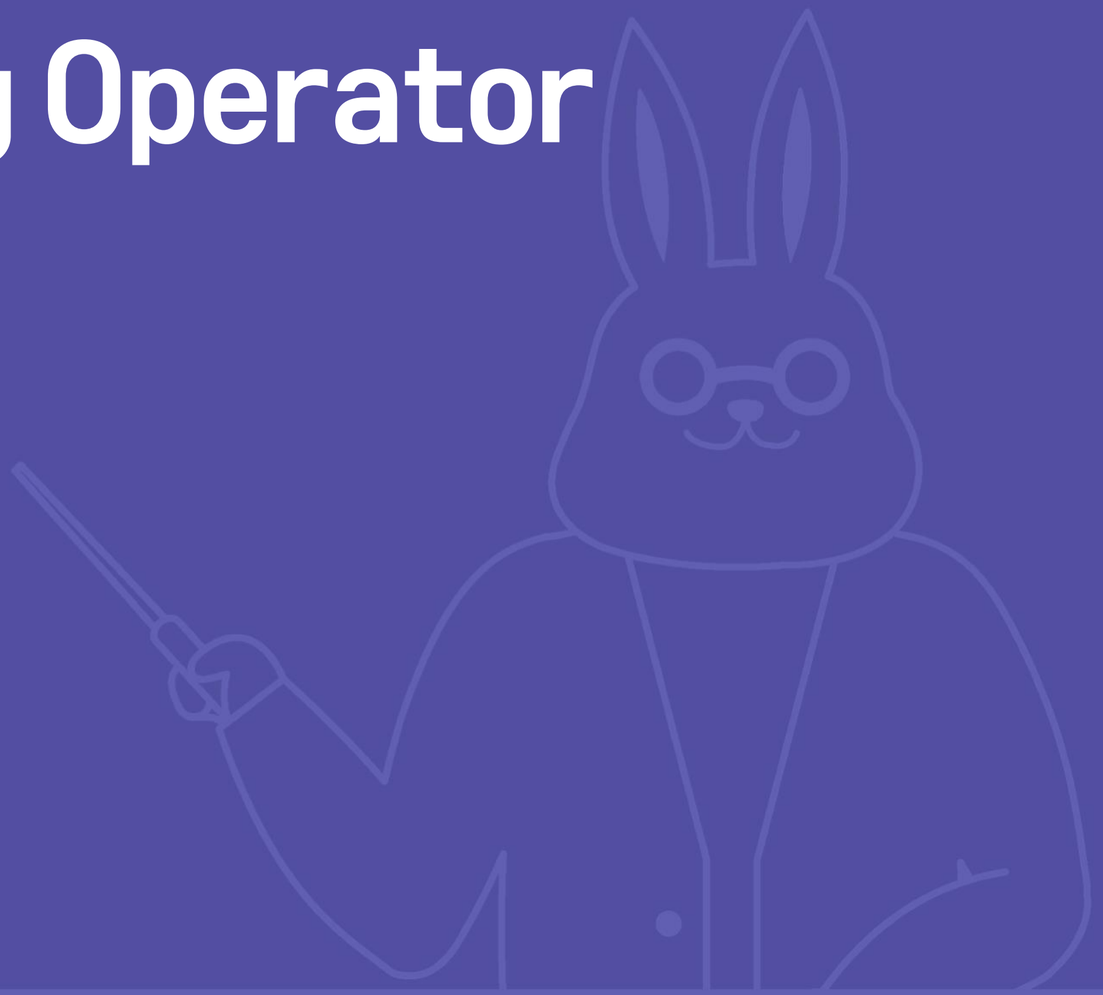
```
type Tail = {
  살랑살랑: () => void;
};

type Human = {
  call: (dogName: string) => void;
};

type Dog = {
  name: string;
  tail?: Tail;
  주인?: Human;
};

function petDog(dog: Dog) {
  // != null: null 또는 undefined가 아닌 경우
  if (dog.주인 != null) {
    dog.주인.call(dog.name);
  }
  if (dog.tail != null) {
    dog.tail.살랑살랑();
  }
}
```

Optional Chaining

```
type Tail = {
  살랑살랑: () => void;
};

type Human = {
  call: (dogName: string) => void;
};

type Dog = {
  name: string;
  tail?: Tail;
  주인?: Human;
};

function petDog(dog: Dog) {
  dog.주인?.call(dog.name);
  dog.tail?.살랑살랑();
}
```

04

Nullish Coalescing Operator



Nullish Coalescing Operator

A ?? B

기존의 A || B는 A가 falsy한 값(0, ‘’, -0, NaN 등)인 경우 B를 반환.
null, undefined를 제외한 falsy 값을 그대로 리턴하고 싶은 경우 사용

es2020에서 추가된 문법이며
좌항이 null, undefined인 경우에만 B를 리턴

Default 값을 정의할 때: ||를 쓰는 경우와 ??를 쓰는 경우

||

```
// price가 0인 경우 -1 반환
function getPrice(product: {
price?: number }) {
  return product.price || -1;
}
```

??

```
// price가 0인 경우 0 반환
function getPrice(product: {
price?: number }) {
  return product.price ?? -1;
}
```

05

Function Overloading

Function Overloading 언제 사용할까? 파라미터만 달라지고, 비슷한 로직이 반복되는 경우.

비슷한 코드의 반복

```
const addZero = (num: string) => (num > 9 ? ‘    ’    : ‘   0’   ) + num

function formatDate(date: Date, format = 'yyyyMMdd'): string {
  const yyyy = date.getFullYear().toString();
  const MM = addZero(date.getMonth() + 1);
  const dd = addZero(date.getDate());

  return format.replace(‘   yyyy', yyyy).replace('MM', MM).replace('dd', dd);
}

function formatDateString(dateStr: string, format = 'yyyyMMdd'): string {
  const date = new Date(dateStr);
  const yyyy = date.getFullYear().toString();
  const MM = addZero(date.getMonth() + 1);
  const dd = addZero(date.getDate());

  return format.replace(‘   yyyy', yyyy).replace('MM', MM).replace('dd', dd);
}

function formatDateTime(datetime: string, format = 'yyyyMMdd'): string {
  const date = new Date(datetime);
  const yyyy = date.getFullYear().toString();
  const MM = addZero(date.getMonth() + 1);
  const dd = addZero(date.getDate());

  return format.replace(‘   yyyy', yyyy).replace('MM', MM).replace('dd', dd);
}
```

파라미터의 타입만 다르고
비슷한 코드가 반복되고 있는 상황.

코드의 중복을 줄이고 재사용성을 높이려면
어떻게 해야 할까?

Function Overloading

파라미터의 형태가 다양한 여러 케이스에 대응하는
같은 이름을 가진 함수를 만드는 것

함수의 다형성(다양한 형태)을 지원하는 것

함수 오버로딩을 위해 해야할 것 2가지

1. 선언 : 함수가 어떤 파라미터 타입들을 다룰 것인지 선언
2. 구현 : 각 파라미터 타입에 대응하는 구체적인 코드를 작성

함수 오버로딩 선언

함수의 이름이 같아야 한다.
매개변수의 순서는 서로 같아야 한다.
반환 타입이 같아야 한다.

Ok

```
class User {
  constructor(private id: string) {}

  setId(id: string): string;
  setId(id: number): string;
}
```

Error

```
class User {
  constructor(private id: string) {}

 // 선언 시에 에러는 나지 않지만 오버로딩 함수 정의 시에 에러
  setId(id: string): void;
  setId(id: string): number; // 반환 타입 다름
  setId(radix: number, id: number): void; // 인수 순서 다름
}
```

함수 오버로딩 구현

매개변수의 개수가 같을 때

```
class User {
  constructor(private id: string) {}

  setId(id: string): void;
  setId(id: number): void;
  setId(id: string | number): void {
    this.id = typeof id === 'number' ?
id.toString() : id;
  }
}
```

매개변수의 개수가 다를 때

```
class User {
  constructor(private id: string) {}

  setId(id: string): void;
  setId(id: number, radix: number):
void;
  setId(id: string | number, radix?:
number): void {
    this.id = typeof id === 'number' ?
id.toString(radix) : id;
  }
}
```

첫 번째 예시를 오버로딩으로 바꿔보자

기존 코드

```
const addZero = (num: string) => (num > 9 ? ‘   ’    : ‘   0’   ) + num

function formatDate(date: Date, format = 'yyyyMMdd'): string {
  const yyyy = date.getFullYear().toString();
  const MM = addZero(date.getMonth() + 1);
  const dd = addZero(date.getDate());

  return format.replace(‘   yyyy', yyyy).replace('MM', MM).replace('dd', dd);
}

function formatDateString(dateStr: string, format = 'yyyyMMdd'): string {
  const date = new Date(dateStr);
  const yyyy = date.getFullYear().toString();
  const MM = addZero(date.getMonth() + 1);
  const dd = addZero(date.getDate());

  return format.replace(‘   yyyy', yyyy).replace('MM', MM).replace('dd', dd);
}

function formatDateTime(datetime: string, format = 'yyyyMMdd'): string {
  const date = new Date(datetime);
  const yyyy = date.getFullYear().toString();
  const MM = addZero(date.getMonth() + 1);
  const dd = addZero(date.getDate());

  return format.replace(‘   yyyy', yyyy).replace('MM', MM).replace('dd', dd);
}
```

함수 오버로딩

```
const addZero = (num: string) => (num > 9 ? ‘   ’    : ‘   0’   )
+ num

function formatDate(date: Date, format = 'yyyyMMdd'): string;
function formatDate(date: number, format = 'yyyyMMdd'): string;
function formatDate(date: string, format = 'yyyyMMdd'): string;
function formatDate(date: string | Date | number, format =
'yyyyMMdd'): string {
  const dateToFormat = new Date(date);

  // ··· dateToFormat validation ···

  const yyyy = dateToFormat.getFullYear().toString();
  const MM = addZero(dateToFormat.getMonth() + 1);
  const dd = addZero(dateToFormat.getDate());

  return format.replace(‘   yyyy', yyyy).replace('MM',
MM).replace('dd', dd);

}
```

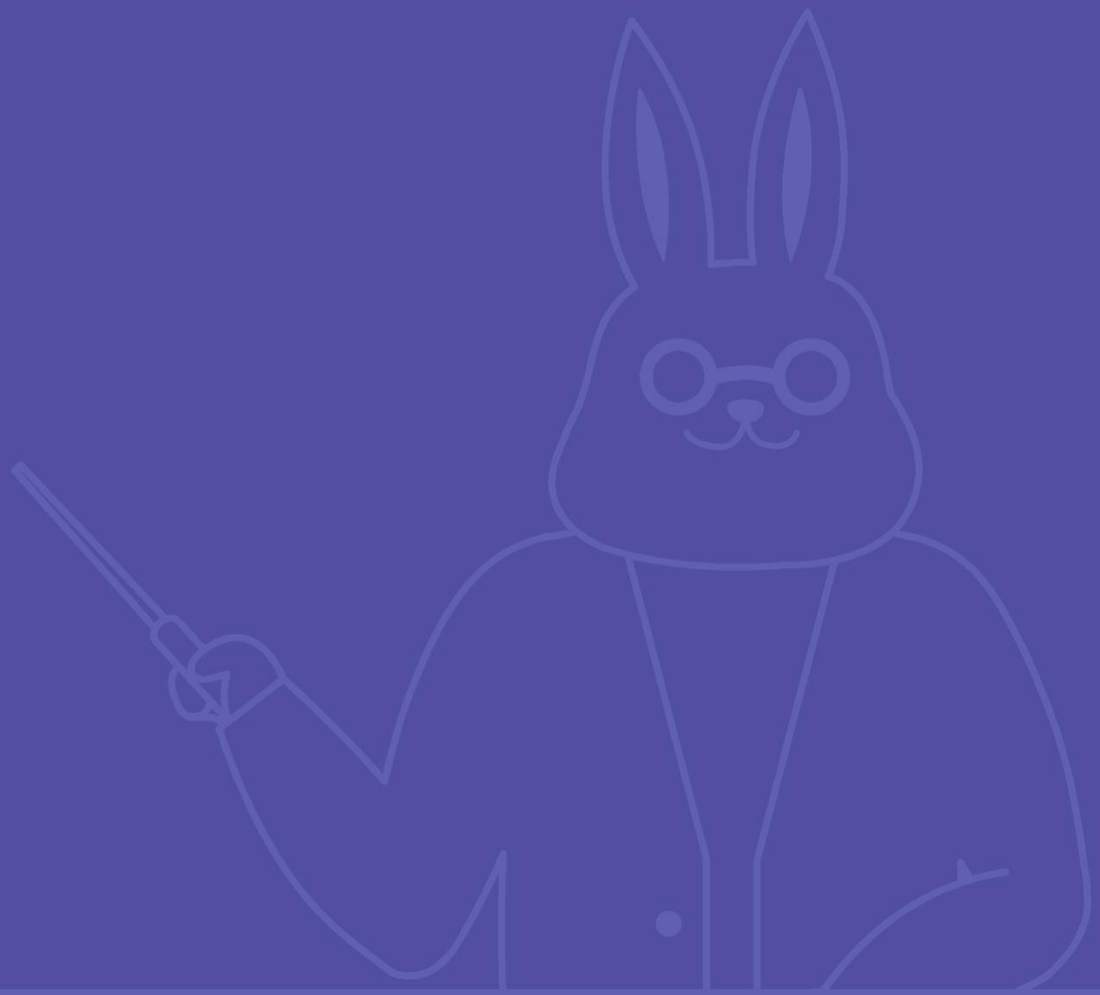
제네릭과의 차이점

타입을 추론할 수 있는 시점과 용도의 범위

	타입 추론 시점	용도의 범위
제네릭	타입이 사용되는 시점	제네릭 타입, 인터페이스, 클래스, 함수, 메서드 등
함수 오버로딩	타입을 선언하는 시점	함수, 메서드

06

Type Assertion



Type Assertion

타입스크립트가 추론하지 못하는 타입을
as keyword를 통해 명시해주는 것

Type Casting과 Type Assertion의 차이

Type casting: 데이터의 타입을 변환
Type assertion: 데이터의 타입을 명시

Type Assertion의 두 가지 형태

As

```
let someValue: unknown = "this is a string";

let strLength: number = (someValue as string).length;
```

꺾쇠(Angle bracket)

```
let someValue: unknown = "this is a string";

let strLength: number = (<string>someValue).length;
```

Type Assertion의 두 가지 형태

As

```
let someValue: unknown = "this is a string";

let strLength: number = (someValue as string).length;
```

꺾쇠(Angle bracket)

```
let someValue: unknown = "this is a string";

let strLength: number = (<string>someValue).length;
```

주의

react의 JSX에서는 꺾쇠괄호를 통한 Type Assertion은 태그와 혼동되기 때문에 잘 사용하지 않습니다.

타입 선언과 타입 단언(Type Assertion)

타입 선언과 단언

```ts
type Duck = {
  꽥꽥: () => void;
  헤엄: () => void;
};

const duck = {} as Duck

const duck: Duck = {
  꽥꽥() {
    console.log('꽥꽥');
  },
  헤엄() {
    console.log('어푸어푸?');
  },
};
```

타입 단언

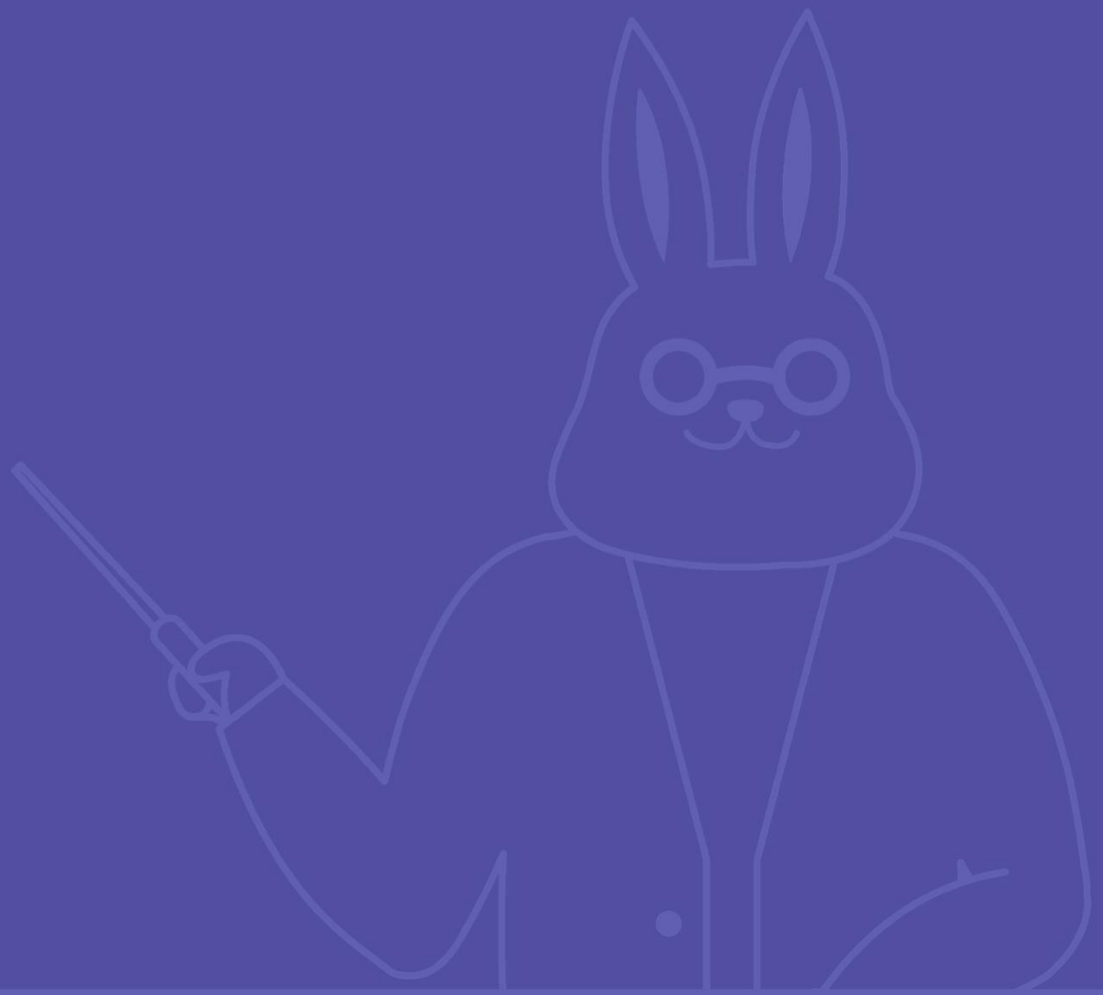
타입스크립트가 개발자의 말만 믿고 Duck 타입으로 인식하여 빈 객체임에도 에러를 뱉지 않음

타입 선언

객체 프로퍼티를 모두 채우도록 강제하기 때문에 실수를 저지를 위험이 낮음

07

Index Signature



Javascript의 Index Signature

Index Signature

```
const dog = {
  breed: 'retriever',
  name: 'elice',
  bark: () => console.log('woof
woof'),
};

dog['breed']
```

객체의 특정 value에 접근할 때 그 value의 key를 **문자열로 인덱싱해 참조**하는 방법

value의 key인 ‘breed’ 문자열로 객체의 value ‘retriever’에 접근

Typescript의 Index Signature

자바스크립트의 인덱스 시그니처에 대한 타입을 지정해주는 것

Index Signature는 언제 사용할까? 객체의 프로퍼티들을 명확히 알 수 없을 때

객체의 프로퍼티가 명확한 경우

```
type Dog = {
  breed: string;
  name: string;
  bark: () => void;
};

const dog: Dog = {
  breed: 'retriever',
  name: 'elice',
  bark: () => console.log('woof woof'),
};

dog['breed'];
```

Index Signature는 언제 사용할까? 객체의 프로퍼티들을 명확히 알 수 없을 때

Index Signature

```
type ArrStr = {
  [key: string]: string | number;
  // [field: string]: string; //
Duplicate index signature for type
'string'
  [index: number]: string;
  length: number;
};
```

- key: 자리 표시 용도. 이름은 어떻게 짓든 상관없음
- ArrStr 타입의 데이터를 인덱스 시그니처로 참조할 때 문자열을 넣으면 string 또는 number 타입 데이터 참조
- 인덱스 시그니처에 number를 넣으면 string 값 참조
- 다른 일반 프로퍼티와 공존 가능
- 하늘 아래 같은 타입의 Index signature는 있을 수 없음
  ex) key가 string인 인덱스 시그니처는 하나만 존재 가능. 한 번 더 정의하려고 하면 타입 에러

Index Signature의 문제점

Index Signature

```
type ArrStr = {
  [key: string]: string | number;
  [index: number]: string;
};

const a: ArrStr = {};

a['str']; // 에러 나지 않음
```

-empty object일 때 인덱스 시그니처로 참조하려 해도 타입 에러가 나지 않음
-key마다 다른 타입을 가질 수 없음 (key 타입이 string 이면 무조건 string 또는 number 타입)
-타입에 유연함을 제공하는 대신 키 이름을 잘못 쓴다든지 하는 휴먼 에러를 저지를 가능성

따라서 어떤 타입이 올지 알 수 있는 상황이라면 정확한 타입을 정의하여 실수를 방지하는 게 좋다.
Index signature는 런타임에 객체의 프로퍼티를 알 수 없는 경우에만 사용할 것.

크레딧

/* elice */

코스 매니저
이재성

콘텐츠 제작자
송현지

강사
송현지

감수자
이재성

디자이너
강혜정

연락처

TEL
070-4633-2015

WEB
https://elice.io

E-MAIL
contact@elice.io