

# Type Design

240602 두선아

이펙티브 타입스크립트, 4장 타입 설계

# Table of Content

28 유효한 상태만 표현하는 타입을 지향하기

29 사용할 때는 너그럽게, 생성할 때는 엄격하게

30 문서에 타입 정보 쓰지 않기

31 타입 주변에 null 값 배치하기

32 유니온의 인터페이스보다는 인터페이스의 유니온을 사용하기



28

유효한 상태만 표현하는 타입을 지향하기

~~Prefer Types That Always  
Represent Valid States~~



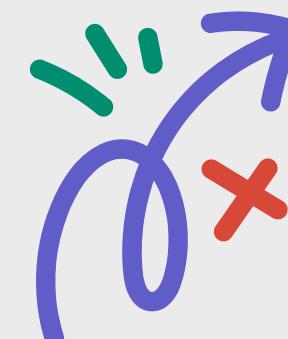
## Prefer Types That Always Represent Valid States

# 유효한 타입 만들기

타입을 만들 때는 **유효한 상태만** 표현할 수 있는 타입을 만드는 것이 중요합니다

👉 유효하지 않은 타입이 존재하는 경우는, 타입 설계가 잘못된 경우입니다

```
async function changePage(state: State, nextPage: string)  
{  
    state.isLoading = true;  
    try {  
        const response = await fetch(getUrlForPage(nextPage));  
        if (!response.ok) {  
            throw new Error(`Unable to load ${nextPage}:  
${response.statusText}`);  
        }  
        const text = await response.text();  
        state.isLoading = false;  
        state.pageText = text;  
    } catch (e) {  
        state.error = "" + e;  
        // isLoading을 false로 바꾸는 로직이 빠져있음  
        // 실행 시, 에러 초기화가 없음  
    }  
}
```



경우	예제
상태 값의 두 가지 속성이 동시에 정보가 부족	요청이 실패한 것인지 여전히 로딩 중인지 알 수 없다
두 가지 속성이 충돌	오류이거나, 로딩 중일 수 있다

state.isLoading

state.error



# Prefer Types That Always Represent Valid States

## 상태 구분 개선

```
// renderPage내 상태 구분 개선
function renderPage(state: State) {
  const { currentPage } = state;
  const requestState = state.requests[currentPage];
  switch (requestState.state) {
    case "pending":
      return `Loading ${currentPage}...`;
    case "error":
      return `Error! Unable to load ${currentPage}: ${requestState.error}`;
    case "ok":
      return `

# ${currentPage}</h1>\n${requestState.pageText}`; } } async function changePage(state: State, nextPage: string) { state.requests[nextPage] = { state: "pending" }; state.currentPage = nextPage; try { const response = await fetch(getUrlForPage(nextPage)); if (!response.ok) { throw new Error(`Unable to load ${nextPage}: ${response.statusText}`); } const pageText = await response.text(); state.requests[nextPage] = { state: "ok", pageText }; } catch (e) { state.requests[nextPage] = { state: "error", error: "" + e }; } }


```

예시: 전자 조종식 항공기의 기장 vs 부기장 스틱 State

- 전자 조종식 스틱의 경우? 스틱의 각도를 평균내는 함수
- 기계적으로 연결되었다면? 상태 표현은 angle만 존재하면 됨

결론: 유효한 타입을 작성할 것

### 상태를 명시적으로 모델링하는 태그된 유니온 사용

```
interface RequestPending {
  state: "pending";
}
interface RequestError {
  state: "error";
  error: string;
}
interface RequestSuccess {
  state: "ok";
  pageText: string;
}
type RequestState = RequestPending | RequestError | RequestSuccess;

interface State {
  currentPage: string;
  requests: { [page: string]: RequestState };
}
```

타입의 유니온

- 타입을 설계할 때, 어떤 값을 포함하고 어떤 값을 제외할 지 신중하게 생각하기
- 유효한 상태 valid state를 표현하는 값만 허용하기



유효한 값과 유효하지 않은 값을  
함께 나타내는 상태값은  
혼란과 에러를 만든다.



코드가 더 길거나 어려워지더라도,  
유효한 값만을 나타내는 코드를 작성하자.

29

사용할 때는 너그럽게, 생성할 때는 엄격하게

*Be Liberal in What You Accept  
and Strict in What You Produce*



Be Liberal in What You Accept and Strict in What You Produce

# 함수 호출을 쉽게 하기

ex) 매개변수를 옵셔널하게 받을 수 있습니다.

내부 로직은? 유니온 타입의 요소별로 코드를 분기할 수 있습니다.

견고함의 원칙 (robustness principle)

존 포스텔 Jon Postel

be conservative in what you do,  
be liberal in what you accept from others

당신의 작업은 엄격하게 하고,  
받아들이는 것은 너그럽게 받아들여야 한다

\*-like는  
자바스크립트의 네이밍 관례입니다



## 예시

```
interface LngLat {  
    lng: number;  
    lat: number;  
}  
type LngLatLike = LngLat | { lon: number; lat: number } | [number, number];  
  
interface Camera {  
    center: LngLat;  
    zoom: number;  
    bearing: number;  
    pitch: number;  
}  
interface CameraOptions extends Omit<Partial<Camera>, "center"> {  
    center?: LngLatLike; // center를 LngLatLike로 변경한 Option 속성  
}  
type LngLatBounds =  
    | { northeast: LngLatLike; southwest: LngLatLike }  
    | [LngLatLike, LngLatLike]  
    | [number, number, number, number];  
// 좋은 설계가 아님  
// 하지만 다양한 타입을 허용해야하는 라이브러리일 수 있다  
  
declare function setCamera(camera: CameraOptions): void;  
declare function viewportForBounds(bounds: LngLatBounds): Camera;
```



Be Liberal in What You Accept and Strict in What You Produce

# 매개변수에는 느슨한 타입, 반환 타입에는 정규형 타입

```
function createUser(data: PartialUser): User {
  return {
    id: generateUniqueId(),
    name: data.name || "Unknown",
    email: data.email || "unknown@example.com",
    age: data.age || 0,
  };
}

function updateUser(id: string, data: PartialUser): User {
  const existingUser = getUserById(id);

  if (!existingUser) {
    throw new Error("User not found");
  }

  return {
    ...existingUser,
    ...data,
  };
}

const newUser = createUser({ name: "Alice" });
console.log(newUser);

const updatedUser = updateUser("123", { email: "alice@example.com" });
console.log(updatedUser);
```

매개변수에는 느슨한 타입  
반환 타입에는 정규형 타입

PartialUser를 매개변수로 받아  
User를 리턴한다



Be Liberal in What You Accept and Strict in What You Produce

# T[] 보다 Iterable<T> 사용하기

```
function processItems<T>(items: Iterable<T>): void {
  for (const item of items) {
    console.log(item);
  }
}

// 배열
const numberArray: number[] = [1, 2, 3, 4, 5];
processItems(numberArray);

// 집합
const numberSet: Set<number> = new Set([1, 2, 3, 4, 5]);
processItems(numberSet);

// 문자열
const text: string = "hello";
processItems(text);

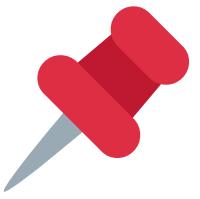
// 맵 (key-value 쌍의 배열을 반복 가능)
const map: Map<string, number> = new Map([
  ["a", 1],
  ["b", 2],
  ["c", 3],
]);
processItems(map);
```

Iterable<T>를 매개변수로 사용하면?

배열 대신

반복 가능한 타입 iterable 을  
받을 수 있습니다.

- Set, Map, String



매개변수와 반환 타입의 재사용을 위해서  
매개변수에는 느슨한 형태  
반환 타입에는 기본 타입을 사용한다.

옵셔널 속성과 유니온 타입은 리턴 타입에는 사용되지 않고  
보통 매개변수로 사용된다



iterable한 매개변수를 위해서  
`T[]` 대신 `Iterable<T>`를 사용할 수 있다.

30

문서에 타입 정보 쓰지 않기

*Don't Repeat Type Information  
in Documentation*



## Don't Repeat Type Information in Documentation

# 타입으로 설명하기



## 예시

```
// 주석으로 설명하는 경우
/** Sort the strings by numeric value (i.e. "2" < "10"). Does not modify
 *  nums. */
function sortNumerically(nums: string[]): string[] {
  return nums.sort((a, b) => Number(a) - Number(b));
}

// 타입으로 설명하는 경우: 규칙을 강제한다
function sortNumerically(nums: readonly string[]): string[] {
  return nums.sort((a, b) => Number(a) - Number(b));
  // ~~~~ ~ ~ Property 'sort' does not exist on 'readonly
  // string[]'
  // 규칙에 맞지 않게 때문에 에러가 발생한다
}
```

## 주석과 타입 구문

구분	동작
주석	구현체와의 정합성이 어긋날 수 있다
타입 구문	타입 체커가 타입 정보를 동기화하도록 강제한다

## 코드와 주석이 맞지 않으면?

둘 다 틀린 것입니다.

## 타입 구문 시스템은?

간결하고, 구체적이며, 쉽게 읽을 수 있다.  
(사용합니다!)

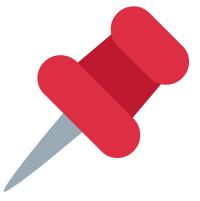
## 특정 매개변수를 설명하기 위해

JSDocs의 @param 구문을 사용할 수 있습니다.

## 타입 정보를 변수명에 사용하지 않도록 합니다.

(단위가 있는 경우를 제외)

변수명	내용
ageNum	변수명을 age로 하고 타입이 number임을 명시하는 것이 좋다
timeMs	변수명이 time일 때는 단위에 대한 정보를 알 수 없다



주석과 태입을 함께 명시하는 경우는?  
중복 정보이거나, 정합성이 어긋난 정보일 수 있다.  
태입 정보에 모순이 발생할 수 있다.



readonly의 경우, 주석으로 명시하지 말고  
태입으로 강제하도록 하자



태입이 명확하지 않은 경우  
단위를 변수명에 포함하는 것을 고려한다

31

타입 주변에 null 값 배치하기

Push Null Values

to the Perimeter of  
Your Types



## Push Null Values to the Perimeter of Your Types

# 타입 주변에 null 값 배치하기

null과 관련된 관계를 가진 결과값을 반환하는 함수입니다.

nums 배열은 비어있을 수 있습니다  
[undefined, undefined]을 반환할 수 있고,  
배열 요소가 0인 경우 잘못된 결과를 반환합니다.

```
// @strictNullChecks: false
function extent(nums: Iterable<number>) {
  // num 배열이 비어있을 수 있음
  let min, max; // undefined, undefined
  for (const num of nums) {
    if (!min) {
      // falsy가 아니라 undefined를 체크해야 한다
      // => 0일 때 덧씌워짐
      min = num;
      max = num;
    } else {
      min = Math.min(min, num);
      max = Math.max(max, num);
    }
  }
  return [min, max];
}
```

null 값 사이의 관계를 이해할 수 있는 코드입니다.

```
function extent(nums: Iterable<number>) {
  let minMax: [number, number] | null = null;
  // 사용하기 더 수월한 코드
  // 타입스크립트가 null 값 사이의 관계를 이해할 수 있도록 했다 (두 값이 다 존재하거나, null임)
  // 단일 객체로 설계를 개선했다
  // null체크가 제대로 동작한다
  for (const num of nums) {
    if (!minMax) { // null이거나!
      minMax = [num, num];
    } else {
      const [oldMin, oldMax] = minMax;
      minMax = [Math.min(num, oldMin), Math.max(num, oldMax)];
    }
  }
  return minMax;
}
...
const [min, max] = extent([0, 1, 2]); // null 단언 자제하자!
const span = max - min; // 정상
...
const range = extent([0, 1, 2]);
if (range) { // null 체크~
  const [min, max] = range;
  const span = max - min; // 정상
}
```



## Push Null Values to the Perimeter of Your Types

# null과 null이 아닌 값

```
interface UserProfile {
  id: number;
  name: string;
  email: string | null; // 이메일이 null일 수 있음
}

// ✖ 사용자 프로필을 비동기적으로 반환하는 async 함수
// Promise를 반환
async function getUserProfileAsync(userId: number): Promise<UserProfile> {
  const users: UserProfile[] = [
    { id: 1, name: "Alice", email: "alice@example.com" },
    { id: 2, name: "Bob", email: "bibim@example.com" },
    { id: 3, name: "Jack", email: null },
  ];

  return users.find((user) => user.id === userId); // undefined 인 것부터 에러
  // {id: -1, name: null, email: null} 같이 초기값 만들면? 복잡해짐
  // `|| null`로 처리하는 것이 코드를 단순히하고, 개발자와 타입 체커가 연관관계를 이해하기 쉽도록 한다.
}

(async function () {
  const userB = await getUserProfileAsync(3);
  if (userB?.email) {
    console.log(`User email: ${userB.email}`);
  } else {
    console.log("User email not available");
  }
})();
```

Using fully non-null APIs and classes  
is preferable to using null values in them.

null 값을 사용하지 않는  
API와 클래스를 사용하는 것이  
일반적으로 더 좋습니다

### 네트워크 요청의 경우?

필요한 데이터가 준비된 다음  
클래스를 만들거나,  
데이터를 가공합니다.

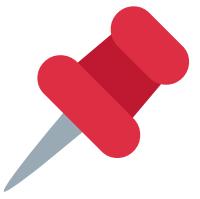
### null 상태에 대한 경우의 수가 많다면?

null 체크가 많아지고,  
버그가 많아집니다.

### Promise와 null 값

null인 경우가 필요한 속성은  
Promise로 바꾸지 않는 것이 좋습니다.

Promise는 데이터를 로드하는 코드를 단순하게 하지만  
null인 경우라면  
데이터를 사용할 때 코드가 복잡해지기도 합니다.



한 값의 null 여부가  
다른 값의 null 여부에  
암시적으로 관련되도록 작성하지 말자



API 작성 시에는, 반환 타입을 큰 객체로 만들어  
전체가 null이거나 null이 아니도록 작성해  
타입을 명료하게 하자.



모든 값이 준비 되었을 때 클래스를 생성하자.

32

유니온의 인터페이스보다는 인터페이스의 유니온을 사용하기

~~Prefer Unions of Interfaces  
to Interfaces with Unions~~



## Prefer Unions of Interfaces to Interfaces with Unions

# 유니온의 인터페이스보다는 인터페이스의 유니온을 사용하기

```
// 😐
// 유니온의 인터페이스 Unions of Interfaces
interface User {
  id: string;
  name: string;
  contact: string | number;
}

const user1: User = {
  id: "1",
  name: "Alice",
  contact: "123-456-7890",
};

const user2: User = {
  id: "2",
  name: "Bob",
  contact: 9876543210,
};
```

contact는  
string 이거나 number입니다

유니온의 인터페이스는  
라이브러리에서 오류가 발생하기 쉽고  
인터페이스를 다루기 어렵습니다



## Prefer Unions of Interfaces to Interfaces with Unions

# 인터페이스의 유니온, 태그된 유니온

```
// 👍  
// 인터페이스의 유니온 Interfaces with Unions  
// 타입 가드를 할 수 있다!  
  
interface Cat {  
    type: "cat";  
    purrs: boolean;  
}  
  
interface Dog {  
    type: "dog";  
    barks: boolean;  
}  
  
type Animal = Cat | Dog;  
  
const animal1: Animal = {  
    type: "cat",  
    purrs: true,  
};  
  
const animal2: Animal = {  
    type: "dog",  
    barks: true,  
};
```

```
// 😊  
interface Layer {  
    layout: FillLayout | LineLayout | PointLayout;  
    paint: FillPaint | LinePaint | PointPaint;  
}  
  
// 👍  
interface FillLayer {  
    layout: FillLayout;  
    paint: FillPaint;  
}  
interface LineLayer {  
    layout: LineLayout;  
    paint: LinePaint;  
}  
interface PointLayer {  
    layout: PointLayout;  
    paint: PointPaint;  
}  
type Layer = FillLayer | LineLayer | PointLayer;
```

분리된 인터페이스를 유니온하면  
잘못된 조합으로 섞이는 것을 방지합니다

### 유용한 경우

- 인터페이스의 속성이 유니온인 경우!
- 여러 개의 선택적 필드가 동시에 값이 있거나  
동시에 undefined인 경우

### 장점

- 타입의 속성들 간 관계를 모델링합니다.
- 타입스크립트가 코드의 정확성을 체크하는 데 도움이 됩니다.



## Prefer Unions of Interfaces to Interfaces with Unions

### 타입의 구조로, 관계를 모델링합니다.

```
// 🚫
interface Person {
  name: string;
  // 타입에 대한 주석은 좋지 않다
  // These will either both be present or not be present
  placeOfBirth?: string; // placeOfBirth와 dateOfBirth의 관계가 표현되지 않았다
  dateOfBirth?: Date;
}

// 👍
interface Person {
  name: string;
  birth?: {
    place: string; // place와 date는 함께 존재한다
    date: Date;
  };
}

function eulogize(person: Person) {
  console.log(person.name);
  const { birth } = person;
  if (birth) {
    // Person 매개변수에서 birth만 체크하면 된다
    console.log(`was born on ${birth.date} in ${birth.place}.`);
  }
}
```

place와 date 속성의 관계

birth 속성이 존재한다면  
place와 date는 함께 존재합니다.

birth 속성이 존재하지 않는다면  
place와 date는 존재하지 않습니다.



유니온 타입의 속성을 여러 개 가지는 인터페이스는  
속성 간의 관계가 분명하지 않다.



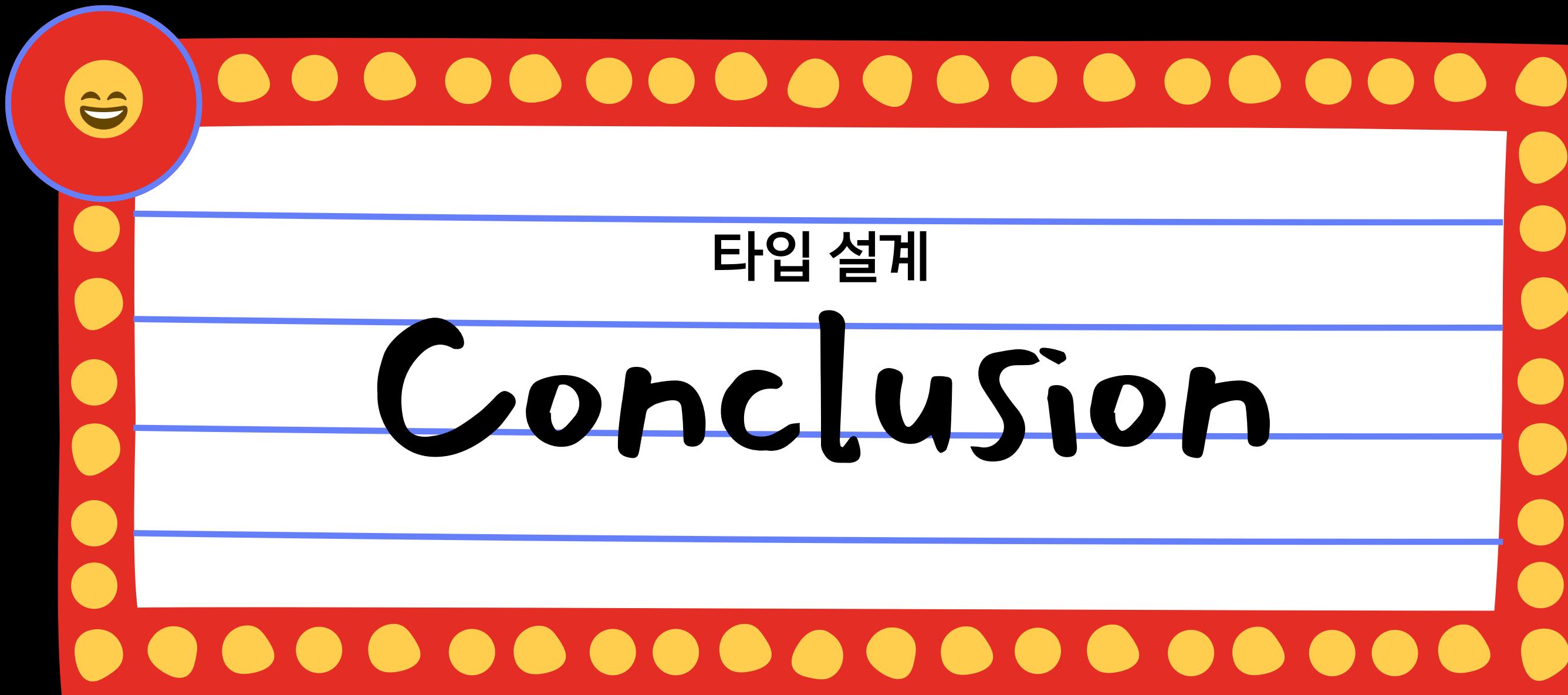
유니온의 인터페이스보다, 인터페이스의 유니온이  
더 정확하고 타입스크립트가 이해하기 좋다

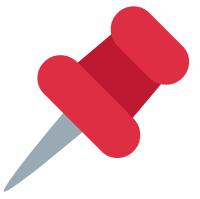


타입스크립트가 제어 흐름을 분석할 수 있도록  
태그를 넣는 것을 고려한다.



옵션 속성이 많으면?  
그룹을 지어서 모델링하는 것을 고려하기





타입 설계를 통해  
속성의 관계를 표현하거나  
readonly를 강제할 수도 있고  
태그로 타입을 구분할 수 있다.

이를 위해서 모순이 없는 유효한 타입을 작성해야 한다.

함수의 매개변수는 느슨한 타입  
반환값은 기본 타입을 사용하자.

반환 타입의 일부 속성이 null이 되는 경우를 방지하기 위해  
모든 API 응답이 끝난 이후 클래스를 만들자.



# Thank you!

이펙티브 타입스크립트, Dan Vanderkam

4장, 타입 추론

예제 코드: <https://github.com/danvk/effective-typescript>