

TYPESCRIPT'S TYPE SYSTEM

타입스크립트의 타입 시스템

240518 두선아

CONTENT

EDITOR	06	EDITOR를 사용하여 타입 시스템 탐색하기
SETS OF VALUES	07	타입이 값들의 집합이라고 생각하기
TYPE SPACE & VALUE SPACE	08	타입 공간과 값 공간의 심벌 구분하기
TYPE ASSERTION	09	타입 단언보다는 타입 선언을 사용하기
OBJECT WRAPPER	10	객체 래퍼 타입 피하기
EXCESS PROPERTY	11	잉여 속성 체크
FUNCTION	12	함수 표현식에 타입 적용하기

06 EDITOR를 사용하여 타입 시스템 탐색하기

Install Typescript

타입스크립트를 설치하면?

- tsc: 타입스크립트 컴파일러
- tsserver: 단독으로 실행할 수 있는 타입스크립트 서버, 언어 서비스 적용

언어 서비스?

TypeScript language services

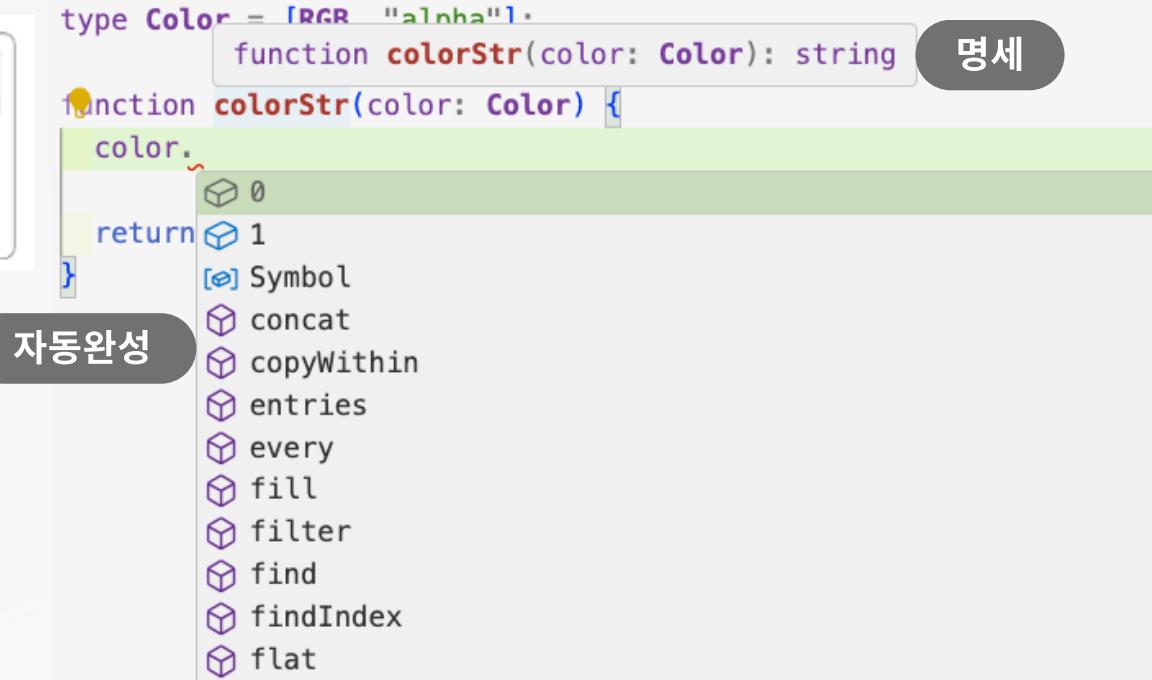
타입 추론

```
let num: number
let num = 10;
```

```
function add(a: number, b: number): number
function add(a: number, b: number) {
    return a + b;
}
```

- 코드 자동 완성 
- 명세 specification 검사
- 검색, 리팩터링
- 타입 추론 정보

자동완성



```
type Color = ["red", "green", "blue", "alpha"];
```

리팩토링 도구

Extract

type alias 분리

Extract to type alias

Move

Move to a new file

Move to file

[Learn more about JS/TS refactorings](#) [Show Disabled](#)

```
type NewType = "green";
```

심볼 이름 변경

```
type Color = [RGB, "alpha"];
```

Color

Enter to Rename, ⌘Enter to Preview

USE YOUR IDE !

06 EDITOR를 사용하여 타입 시스템 탐색하기

값의 타입 변화

타입스크립트가 속성을 어떻게 추론하는지 살펴보기

- 조건문 분기에서 값의 타입 변화 (타입 좁히기, 넓히기)
- 연속된 메서드 호출에서 추론된 제너릭 타입을 조사

```
const arraySample = [0, "1", 2, 3];

const sortedNumArray = arraySample
  .map((e) => Number(e))
  .filter((e) => !isNaN(e))
  .sort((a, b) => a - b);
```

연속된 메서드 호출

```
(method) Array<string | number>.map<number>(callbackfn: (value: string | number, index: number, array: (string | number)[]) => number, thisArg?: any): number[]
```

```
(method) Array<number>.filter(predicate: (value: number, index: number, array: number[]) => unknown, thisArg?: any): number[] (+1 overload)
```

```
(method) Array<number>.sort(compareFn?: ((a: number, b: number) => number) | undefined): number[]
```

Go To Definition

- 라이브러리, 라이브러리의 타입 선언을 탐색할 때 도움이 됨
- d.ts 파일

```
const sortedNumArray = arraySample
  .map((e) => Number(e))
  .filter((e) => !isNaN(e))
```

타입 탐색하기

lib.es5.d.ts

```
1380 * ``ts
1381 * [11,2,22,1].sort((a, b) => a - b)
1382 *
1383 */
1384 sort(compareFn?: (a: T, b: T) => number): this;
1385 /**
1386 * Removes elements from an array and, if necessary, inserts new elements in their
1387 * place, returning the deleted elements.
1388 * @param start The zero-based location in the array from which to start removing
1389 * elements.
1390 * @param deleteCount The number of elements to remove.
1391 * @returns An array containing the elements that were deleted.
1392 */
```

USE YOUR EDITOR TO INTERROGATE AND EXPLORE THE TYPE SYSTEM

06 EDITOR를 사용하여 타입 시스템 탐색하기

Things to Remember

1

타입스크립트 언어 서비스를 제공하는 편집기에서 타입스크립트 언어를 활용

Take advantage of the TypeScript language services by using an editor that supports them

2

편집기를 사용해서 어떻게 타입 시스템이 동작하는지,
타입스크립트가 어떻게 타입을 추론하는지 확인

Use your editor to build an intuition for how the type system works and how TypeScript infers types.

3

타입스크립트가 동작을 어떻게 모델링하는지 알기 위해
타입 선언 파일을 찾아보자

Know how to jump into type declaration files to see how they model behavior.

4

타입스크립트 리팩토링 도구를 활용하자

Familiarize yourself with TypeScript's refactoring tools, e.g., renaming symbols and files.

USE YOUR IDE !

07 타입이 값들의 집합이라고 생각하기

Type?

런타임에 모든 변수는 고유한 값을 가집니다
ts가 오류 체크를 하는 순간에는? 값이 아닌 타입을 가지고 있습니다

- 타입은? 할당 가능한 값들의 집합, sets of values
- 이 집합은? 타입의 범위, type's domain

- 42
- null
- undefined
- 'Canada'
- {animal: 'Whale', weight_lbs: 40_000}
- /regex/
- new HTMLButtonElement
- (x, y) => x + y

집합 set

strictNullCheck

strictNullCheck 설정에 따른 집합 차이
예를 들어 number 타입은 Number, null, undefined를 포함할 수 있습니다.

공집합 empty set

가장 작은 집합, never는 아무 값도 포함하지 않습니다

```
const x: never = 12;
// ~ Type 'number' is not assignable to type 'never'.
```

유닛 타입 unit type

literal 타입, 한 가지 값만 가지는 집합입니다

```
type A = "A";
type B = "B";
type Twelve = 12;
```

합집합 union

값 집합이 둘 이상 조합된 타입입니다

```
type AB = "A" | "B";
type AB12 = "A" | "B" | 12;
```

07 타입이 값들의 집합이라고 생각하기

집합의 관점에서의 타입

assignable

- "할당 가능한", assignable
- ~의 원소(값과 타입) member
 - ~의 부분집합(타입과 타입) subset

subset

집합 관점에서 타입체커의 역할은? 하나의 집합이 다른 집합의 **부분 집합**인지 확인

```
// OK, {"A", "B"} is a subset of {"A", "B"}:
const ab: AB = Math.random() < 0.5 ? "A" : "B";
const ab12: AB12 = ab; // OK, {"A", "B"} is a subset of {"A", "B", 12}

declare let twelve: AB12;
const back: AB = twelve;
//     ~~~~ Type 'AB12' is not assignable to type 'AB'
//             Type '12' is not assignable to type 'AB'
```

실제 다루게 되는 타입은 대부분 범위가 무한대입니다

```
interface Identified {
  id: string;
}
// 어떤 객체가 string으로 할당 가능한 id 속성을 가지고 있다면? 객체는 Identified다. (구조적 타이핑)
// excess property checking을 생각하다보면 간과하기 쉬움 (Item 11에 나올 것)
```

07 타입이 값들의 집합이라고 생각하기

집합의 연산

union

| 연산자는 타입의 합집합 union을 연산합니다

intersection

& 연산자는 타입의 교집합을 연산합니다

```
interface Person {
  name: string;
}
interface Lifespan {
  birth: Date;
  death?: Date;
}
type PersonSpan = Person & Lifespan;
// Person과 Lifespan 인터페이스는 공통으로 가지는 속성이 없다
// 타입 연산자는 인터페이스의 속성이 아닌, 값의 집합에 적용된다. (타입의 범위에 적용된다)
// - Person과 Lifespan을 둘 다 가지는 값은 intersection에 속하게 된다
// - 세 가지보다 더 많은 속성을 가지는 값도 PersonSpan에 속한다.
```

합집합과 교집합?

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

type ABInteraction = A & B;
type KeysOfABInteraction = keyof ABInteraction; // "a" | "b" | "c"

type ABUnion = A | B;
type KeysOfAUnion = keyof ABUnion; // "b"
```

// 집합과 타입 연산

```
keyof (A | B) = (keyof A) & (keyof B)
// 교집합에 대한 keyof
// A와 B의 공통 키만을 포함하는 인터섹션 타입
keyof (A & B) = (keyof A) | (keyof B)
// 합집합에 대한 keyof
// A와 B의 모든 키를 포함하는 유니언 타입
```



07 타입이 값들의 집합이라고 생각하기

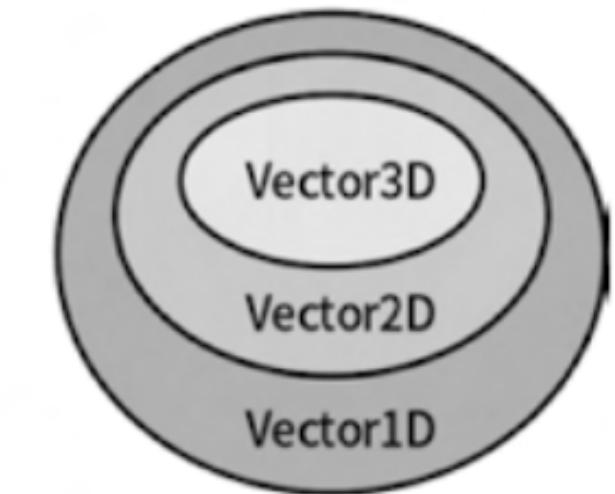
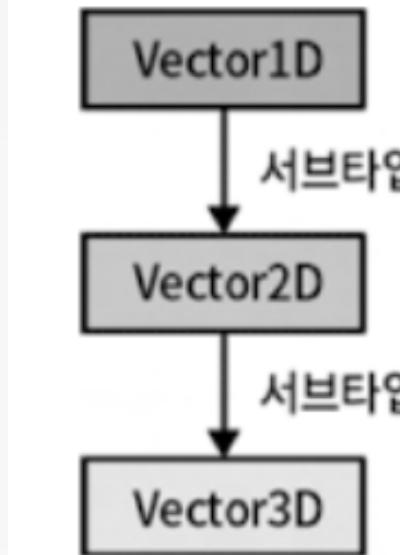
확장 Extends

타입의 집합이라는 관점에서 extends의 의미는 **assignable** (~에 할당 가능한)과 비슷하게 **subset** (~의 부분 집합)이라는 의미로 받아들일 수 있습니다

subtype 서브타입

- 어떤 집합(set)이 다른 집합(set)의 부분 집합(subset)이라는 뜻
- 클래스 관점에서는 서브클래스
- 집합의 관점에서는, 상속 관계가 아니라, 벤 다이어그램으로 그리는 것이 적절합니다

```
interface Vector1D {
  x: number;
}
interface Vector2D extends Vector1D {
  // Vector1D의 서브 타입이다
  y: number;
}
interface Vector3D extends Vector2D {
  // Vector2D의 서브 타입이다
  z: number;
}
```



07 타입이 값들의 집합이라고 생각하기

확장 Extends

제네릭 generic

- 제네릭 타입에서 한정자 (~의 **subset**) 로도 쓰입니다.

```
function getKey<K extends string>(val: any, key: K) {
    // 객체 상속의 관점이라면? 객체 Wrapper 타입 String의 서브클래스를 정의하는 것이지만...
    // 집합의 관점에서 생각한다면? string의 부분 집합 범위를 가지는 타입이 된다. K & string
    // ...
}
```

- 예시: 객체의 키 타입을 반환하는 **keyof T**를 사용

```
function sortBy<K extends keyof T, T>(vals: T[], key: K): T[] {
    // ...
}
```

07 타입이 값들의 집합이라고 생각하기

배열과 튜플

배열과 튜플

- 배열 number[]을 튜플 [number, number]에 할당할 수 없습니다
- 튜플 [number, number]를 배열 number[]에 할당할 수 있습니다

subset 이해하기

```
const list = [1, 2];
//   ^? const list: number[]
const tuple: [number, number] = list;
// ~~~~~ Type 'number[]' is not assignable to type '[number, number]'
//           Target requires 2 element(s) but source may have fewer
```

타입이 값이라는 것은?

- 타입이 값의 집합이라는 건? 동일한 값의 집합을 가지는 두 타입은 같습니다
- 같은 타입을 두 번 정의할 필요는 없습니다

집합 관점의 용어 정리

타입스크립트 용어	집합 용어
never	\emptyset (공집합)
리터럴 타입	원소가 1개인 집합
값이 T에 할당 가능	값 $\in T$ (값이 T의 원소)
T1 T2에 할당 가능	$T_1 \subseteq T_2$ (T_1 이 T_2 의 부분 집합)
T1 T2를 상속	$T_1 \subseteq T_2$ (T_1 이 T_2 의 부분 집합)
$T_1 \mid T_2$ (T_1 과 T_2 의 유니온)	$T_1 \cup T_2$ (T_1 과 T_2 의 합집합)
$T_1 \& T_2$ (T_1 와 T_2 의 인터섹션)	$T_1 \cap T_2$ (T_1 과 T_2 의 교집합)
unknown	전체(universal) 집합

07 타입이 값들의 집합이라고 생각하기

Think of types as sets of values (the type's domain). These sets can either be finite (e.g., boolean or literal types) or infinite (e.g., number or string).

Things to Remember

1

타입을 값의 집합으로 이해하기 (타입의 범위)

집합은 유한하거나, 무한하다

TypeScript types form intersecting sets (a Venn diagram) rather than a strict hierarchy.

Two types can overlap without either being a subtype of the other.

2

타입스크립트 타입은 엄격한 상속 관계가 아니라, 겹쳐지는 집합으로 표현된다

두 타입은 서브타입이 아니면서도 겹쳐질 수 있다

Remember that an object can still belong to a type even if it has additional properties that were not mentioned in the type declaration.

3

객체에 타입 선언에 존재하지 않는 추가 속성이 있더라도, 그 타입에 속할 수 있다

Type operations apply to a set's domain. The domain of $A \mid B$ is the union of the domains of A and B.

4

타입 연산은 집합의 범위에 적용된다.

- A or B는 A범위와 B범위의 합집합이다.
- A and B는 A범위이거나, B범위이다.

Think of "extends," "assignable to," and "subtype of" as synonyms for "subset of."

5

상속, 할당가능, 서브타입에 속한다는 것은? 부분 집합에 속한다는 것이다

08 타입 공간과 값 공간의 심벌 구분하기

symbol

symbol은 타입 공간(type space)이나 값 공간(value space) 중 한 곳에 위치합니다

- 이름이 같더라도, 속하는 공간에 따라 다른 것을 나타낼 수 있습니다!

타입 공간과 값 공간의 심벌

```
interface Cylinder {
    radius: number;
    height: number;
} // Cylinder는 타입. Type space

const Cylinder = (radius: number, height: number) => ({ radius, height }); // 
Cylinder는 함수. Value space

// 오류 상황
function calculateVolume(shape: unknown) {
    if (shape instanceof Cylinder) {
        shape.radius;
        // ~~~~~ Property 'radius' does not exist on type '{}'
        // 런타임에 Cylinder 심볼을 값으로 연산하려 했으므로, 값 공간의 함수 Cylinder를 참조한다
    }
}
```

- symbol이 value인지 type인지는 문맥을 살펴 알아내야 합니다

문맥 살피기

```
interface Person {
    first: string;
    last: string;
}
const jane: Person = { first: "Jane", last: "Jacobs" };
//   —————— Values
//   —————— Type

function email(to: Person, subject: string, body: string): Response {
    //   —————— Values
    //   —————— Types
    // ...
}
```

08 타입 공간과 값 공간의 심벌 구분하기

type vs value

- typescript playground를 활용하면
컴파일된 자바스크립트 결과물을 함께 보면서
type space & value space를 확인할 수 있습니다

type space

```
// ✅ Type
interface A { a: number; b: string }
interface B { b: string; c: boolean }

type ABInteraction = A & B;
type KeysOfABInteraction = keyof ABInteraction; // "a" | "b" | "c"

type ABUnion = A | B;
type KeysOfABUnion = keyof ABUnion; // "b"

// ✅ Value
const valueA = {
  a: 100,
  b: "hello~"
} as A
let valueB:B= {
  b: "hello",
  c: true
}
```

type, interface는 type

자바스크립트 컴파일 후 제거되면 type 정보

타입 선언 또는 단언문 다음에 나오는 심벌은 type

value space

```
// ✅ Value
const valueA = {
  a: 100,
  b: "hello~"
};

let valueB = {
  b: "hello",
  c: true
};
```

const, let, var는 value
자바스크립트 컴파일 후 남으면 value 정보
= 다음에 나오는 심벌은 value

class와 enum은 상황에 따라 type 또는 value

TYPE SPACE vs VALUE SPACE

08 타입 공간과 값 공간의 심벌 구분하기

class

클래스가 타입으로 쓰일 때는
형태(속성과 메서드)가 사용됨

클래스가 값으로 쓰일 때는
생성자가 사용됨

value

type

```
class Cylinder {
    radius: number;
    height: number;
    constructor(radius: number, height: number) {
        this.radius = radius;
        this.height = height;
    }
}

function calculateVolume(shape: unknown) {
    if (shape instanceof Cylinder) {
        // Cylinder를 값으로 연산
        // 생성자가 사용됨
        shape;
        // ^? (parameter) shape: Cylinder, 타입으로 참조
        // 형태(속성과 메서드)가 사용됨
        shape.radius;
        // ^? (property) Cylinder.radius: number, 타입으로 참조
    }
}
```

KNOW HOW TO TELL WHETHER A SYMBOL IS IN THE TYPE SPACE OR VALUE SPACE

08 타입 공간과 값 공간의 심벌 구분하기

typeof

- type space(타입 공간)의 `typeof`은 subset으로 사용할 수 있고 `type` 구문으로 이름을 붙이는 용도로 사용할 수 있습니다
- value space에서, `typeof`은 자바스크립트 런타임의 `typeof`입니다.

```
interface Person {  
    first: string;  
    last: string;  
}  
  
const jane: Person = { first: "Jane", last: "Jacobs" };  
const email = (to: Person, subject: string, body: string) => {};  
  
type T1 = typeof jane;  
// ^? type T1 = Person  
type T2 = typeof email;  
// ^? type T2 = (to: Person, subject: string, body: string) => Response  
  
const v1 = typeof jane; // Value is "object"  
const v2 = typeof email; // Value is "function"  
  
console.log(v1, v2); // "object", "function"
```

interface와 typeof

```
class Cylinder {  
    radius: number;  
    height: number;  
    constructor(radius: number, height: number) {  
        this.radius = radius;  
        this.height = height;  
    }  
}
```

class와 typeof

```
const v = typeof Cylinder; // Value is "function"  
// 클래스가 자바스크립트에서는 함수로 구현되기 때문에, v는 function이다.  
type T = typeof Cylinder; // Type is typeof Cylinder  
// T의 Cylinder는 인스턴스의 타입이 아니라, 생성자 함수이다.  
// T를 value space에서는 확인할 수 없음. (타입이기 때문)  
  
console.log(v); // "function"
```

InstanceType

InstanceType 제너릭을 사용해
생성자 타입과 인스턴스 타입 전환

```
declare let fnA: T; // 타입이 생성자 typeof Cylinder임을 확인 가능(fn is not defined)  
let fnB: T = Cylinder; // 선언과 함께 값을 할당해주면 함수를 실행할 수 있음  
type fnC = InstanceType<typeof Cylinder>; // 타입이 생성자 typeof Cylinder임을 확인 가능(fn is not defined)  
  
const c = new fnB(10, 5);  
console.log(v, c);
```

08 타입 공간과 값 공간의 심벌 구분하기

속성 접근자 []

property accessor

- obj['field']와 obj.field의 값은 동일하더라도, 타입은 다를 수 있습니다
- 타입의 속성에 접근할 때는, []를 사용합니다

```
const jane: Person = { first: "Jane", last: "Jacobs" };
const first: Person["first"] = jane["first"]; // Or jane.first
// _____ Values. "Jane"
// _____ Types, 타입 맥락에서 쓰였기 때문에 타입
```

- 인덱스에는 union, 기본형 (js 타입)을 포함해 어떤 타입이든 사용할 수 있습니다

```
type PersonEl = Person["first" | "last"]; // union!
// ^? type PersonEl = string
type Tuple = [string, number, Date]; // 기본형
type TupleEl = Tuple[number];
// ^? type TupleEl = string | number | Date
```

class

특정 객체를 생성하기 위한 blueprint

namespace

네임스페이스를 통해
함수, 클래스, 변수 등에 접근

코드를 논리적으로 그룹화하고
전역 스코프의 오염을 방지

access error 메시지

Cannot access 'Person.first' because 'Person' is a type, but not a namespace. Did you mean to retrieve the type of the property 'first' in 'Person' with 'Person["first"]'?
 Person = { first: 'Jane', last: 'Jacobs' }:
 - Cannot access 'Person.first' because 'Person' is a type, but not a namespace. Did you mean to retrieve the type of the property 'first' in 'Person' with 'Person["first"]'? (2713)
 = type Person.first = /*unresolved*/ any
 = View Problem (⌘F8) Quick Fix... (⌘.)

08 타입 공간과 값 공간의 심벌 구분하기

Value vs Type

this

값 value 자바스크립트의 this

타입 type 다형성 this

value로 쓰이는 this는?
JS의 this입니다

type으로 쓰이는 this는?
다형성 this입니다

polymorphic this는? 자바스크립트와 마찬가지로 현재 실행 컨텍스트를 참조합니다.
여러 클래스나 객체에서 공통된 메서드를 정의하면서도 각기 다른 동작을 구현할 수 있는데,
주로 상속과 인터페이스를 통해 구현됩니다. 유연성을 활용!

polymorphic this: 클래스와 상속, 오버라이딩

```
class Animal {
  move(distance: number) {
    console.log(`[${this.constructor.name}] moved ${distance} meters.`);
    // this는 메서드를 호출한 객체이다
  }
}

class Dog extends Animal {
  move(distance: number) {
    console.log("Dog is running...");
    super.move(distance);
  }
}

class Bird extends Animal {
  move(distance: number) {
    console.log("Bird is flying...");
    super.move(distance);
  }
}

const dog = new Dog();
dog.move(10); // Dog is running... Dog moved 10 meters.

const bird = new Bird();
bird.move(20); // Bird is flying... Bird moved 20 meters.
```

polymorphic this: 인터페이스, 메서드 체이닝

```
interface Movable {
  move(distance: number): this;
}

class Car implements Movable {
  move(distance: number): this {
    console.log(`Car drove ${distance} kilometers.`);
    return this;
  }

  refuel(amount: number): this {
    console.log(`Car refueled with ${amount} liters.`);
    return this;
  }
}

class Plane implements Movable {
  move(distance: number): this {
    console.log(`Plane flew ${distance} miles.`);
    return this;
  }

  maintain(): this {
    console.log("Plane underwent maintenance.");
    return this;
  }
}

const car = new Car();
const plane = new Plane();
```

08 타입 공간과 값 공간의 심벌 구분하기

Value vs Type

extends

서브클래스 또는 서브타입, 제네릭 타입의 한정자

서브 클래스

```
class Animal {
  move() {
    console.log("Moving along!");
  }
}

class Dog extends Animal {
  bark() {
    console.log("Woof! Woof!");
  }
}
```

제네릭 타입의 한정자

```
function logLength<T extends { length: number }>(item: T): void {
  console.log(item.length);
}

logLength("Hello, TypeScript!"); // 18
logLength([1, 2, 3, 4, 5]); // 5

logLength(123); // Error: Argument of type 'number' is not assignable to parameter of type '{ length: number; }'.
```

```
interface Person {
  name: string;
  age: number;
}

interface Employee extends Person {
  employeeId: number;
}

const employee: Employee = {
  name: "John",
  age: 30,
  employeeId: 1234,
};
```

서브 타입

&와 |

	&	
값 value	AND	OR
타입 type	intersection	union

const

	const
값 value	const 새 변수 선언
타입 type	as const 리터럴 표현식의 추론된 타입을 바꿈

08 타입 공간과 값 공간의 심벌 구분하기

Value vs Type

in loop 또는 mapped 타입

```
const obj = { a: 1, b: 2, c: 3 };

for (const key in obj) {
  console.log(`#${key}: ${obj[key]}`);
}

// 출력:
// a: 1
// b: 2
// c: 3
```

loop

```
type Optional<T> = {
  [K in keyof T]?: T[K];
  // in 연산자는 keyof T를 통해 Person의 모든 속성을 반복
  // 각 속성에 대해 ?:를 사용하여 옵셔널
};
```

mapped

```
interface Person {
  name: string;
  age: number;
}
```

```
type OptionalPerson = Optional<Person>;
// { name?: string; age?: number; }
```

반복문에서 객체의 속성을 순회할 때 사용됩니다.
 주로 for...in 루프와 함께 사용되며
 객체의 각 속성을 반복하면서 작업을 수행합니다.

Mapped 타입에서 in 연산자는 조건부 타입과 함께 사용될 때
 타입 변환을 수행하는 데 사용됩니다
 (객체의 모든 속성을 반복하면서 새로운 타입을 만들거나 수정하는 데 사용)

KNOW HOW TO TELL WHETHER A SYMBOL IS IN THE TYPE SPACE OR VALUE SPACE

08 타입 공간과 값 공간의 심벌 구분하기

Know how to tell whether you're in type space or value space while reading a TypeScript expression.
Use the TypeScript playground to build an intuition for this.

Things to Remember

타입스크립트가 잘 동작하지 않는다면?
type space과 value space를 혼동했을 가능성이 크다!

1 타입 공간과 값 공간을 구분하는 방법을 터득해야함

타입스크립트 플레이그라운드 <https://www.typescriptlang.org/play/?#code/>

Every value has a static type, but this is only accessible in type space.

Type space constructs such as type and interface are erased and are not accessible in value space.

2 모든 값은 타입을 가지지만, 타입은 타입 공간에서만 접근할 수 있다.

타입 공간에서 만든 타입, 인터페이스는 값 공간에서는 삭제된다.

type과 interface는 type space에 존재

Some constructs, such as class or enum, introduce both a type and a value

3 class와 enum은 타입과 값으로 둘 다 사용될 수 있다

typeof, this, and many other operators and keywords have different meanings in type space and value space

4 typeof, this 등의 많은 연산자와 키워드는 type space와 value space에서 다른 목적으로 사용된다

09 타입 단언보다는 타입 선언을 사용하기

Why?

- 타입 단언은 타입 체커에게 오류를 무시하라고 하는 것이기 때문입니다

```
interface Person {
  name: string;
}

// Type Annotations은 할당된 값이 해당 인터페이스를 만족하는지
const alice: Person = { name: "Alice" };
// ^? const alice: Person

// Type Assertions는 추론한 타입이 있더라도, 해당 타입으로 간주한다
const bob = { name: "Bob" } as Person; // Type Assertions
// ^? const bob: Person
```

- 잉여 속성 체크 excess property checking도 적용되지 않습니다

```
const alice: Person = {
  name: "Alice",
  occupation: "TypeScript developer",
  // ~~~~~ Object literal may only specify known properties,
  //           and 'occupation' does not exist in type 'Person'
};
const bob = {
  name: "Bob",
  occupation: "JavaScript developer",
} as Person; // 에러가 없음!
```

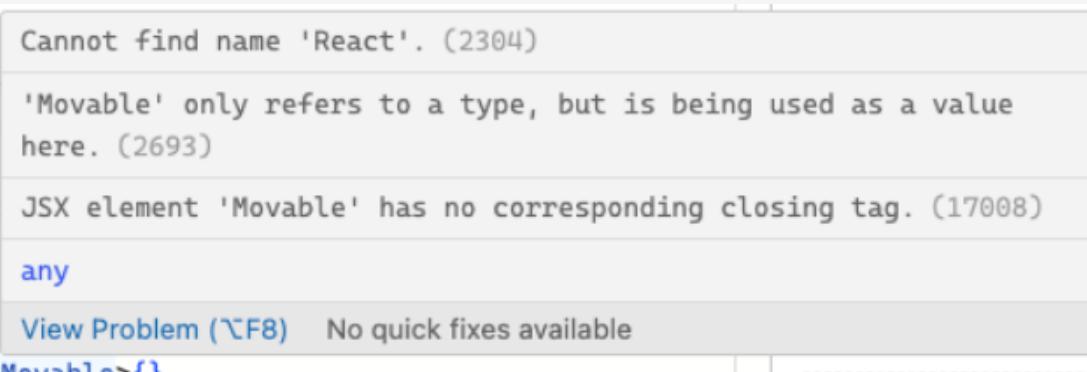
09 타입 단언보다는 타입 선언을 사용하기

= <타입>{ }?

- 이전에 타입 단언을 사용하던 코드 형태입니다

```
const bob = <Person>{};
```

- 현재는 tsx의 컴포넌트 태그로 인식되기 때문에 쓰이지 않습니다



화살표 함수

- 최종적으로 원하는 타입을 직접 명시하고, 타입스크립트가 할당문의 유효성을 검사하도록 합니다
- 단, 함수 호출 체이닝이 연속되는 곳은, 체이닝 시작부터 명명된 타입을 가져야 합니다 (정확한 곳에 오류를 표기하기 위함!)

런타임 오류 주의!

```
interface Person {
  name: string;
}
// 타입 단언
const people = ["alice", "bob", "jan"].map((name) => ({ name } as Person)); // Type is Person[]

// 단언 시 런타임 오류 발생
const wrongPeople = ["alice", "bob", "jan"].map((name) => ({} as Person)); // Type is Person[]

// 최종적으로 원하는 타입을 직접 명시하고, 타입스크립트가 할당문의 유효성을 검사하도록 함
const goodPeople: Person[] = ["alice", "bob", "jan"].map((name) => ({ name })); // OK
```

09 타입 단언보다는 타입 선언을 사용하기

Type Assertion이 필요한 경우

- 타입 체커가 추론한 타입보다, 개발자가 판단한 타입이 더 정확할 때 사용합니다. ex) DOM element

```
document.querySelector("#myButton")?.addEventListener("click", (e) => {
  e.currentTarget;
  // ^? (property) Event.currentTarget: EventTarget | null
  // currentTarget is #myButton is a button element: 타입스크립트가 알지 못하는 정보, why? DOM에 접근할 수 없기 때문에
  const button = e.currentTarget as HTMLButtonElement;
  //   ^? const button: HTMLButtonElement
});
```

nonNullAssertion

변수의 접두사로 쓰인 !는 boolean의 부정문입니다
 접미사로 쓰인 !는 그 값이 Null이 아니라는 단언문으로 해석됩니다

주의!

- 단언문은 컴파일 과정에서 제거됩니다
- 타입 체커는 알지 못하지만, 값이 null이 아님을 확신할 수 있을 때 사용합니다
- 확신할 수 없다면 조건문으로 null 체크 진행합니다

타입 변환과 unknown

타입이 서로의 서브 타입이 아니면 타입 변환이 불가능합니다

- 하지만 모든 타입은 unknown의 서브타입이기 때문에, unknown 단언은 항상 동작합니다

만약 unknown 타입을 사용해 임의의 타입 간의 변환이 가능한, 항상 동작하는 단언을 사용하는 경우?

- 무언가 위험한 동작을 하고 있음을 알 수 있습니다

```
const el = document.body as unknown as Person; // OK
```



TYPE ASSERTIONS ?

09 타입 단언보다는 타입 선언을 사용하기

Things to Remember

1

타입 단언보다 타입 선언을 사용하자

Prefer type annotations (`: Type`) to type assertions (`as Type`).

2

화살표 함수의 반환 타입을 명시하는 법을 알자

Know how to annotate the return type of an arrow function.

3

타입스크립트보다 더 잘 알고 있는 게 확실한 상황에서만 타입 단언과 non-null 단언을 쓰자

Use type assertions and non-null assertions only when you know something about types that TypeScript does not.

4

협업을 위해 타입 단언에 대한 주석을 남기자

When you use a type assertion, include a comment explaining why it's valid.

10 객체 래퍼 타입 피하기

Primitive Type & Object Wapper Type

string의 property에 접근하려 할 때

```
"primitive".charAt(3); // 'm'
```

기본형 값의 7타입: string, number, boolean, null, undefined, symbol(ES2015), bigint(최종 확정 단계)
(immutable이며 메서드를 가지지 않습니다)

래퍼 객체: String, Number, Boolean, Symbol, BigInt

stirng을 String 객체로 wrapping하고
(coerce)

래퍼 객체에서 메서드를 호출하고
(new String(string), String의 메서드를 상속하고, 속성 참조)

래핑한 객체를 버립니다.
(property가 resolved되었을 때)

특징

- 기본형에 속성을 추가한다면? 래퍼 객체와 함께 버려집니다
- 타입스크립트가 제공하는 타입 선언은 기본형 타입입니다
- string은 String에 할당할 수 있습니다 (String은 string에 할당할 수 없습니다)

```
const s: String = "primitive"; // 할당 가능하지만, 기본형 타입을 사용하자
const n: Number = 12;
const b: Boolean = true;

// new 키워드 없이 BigInt와 Symbol을 호출하면 기본형을 생성하기 때문에 사용해도 됨
typeof BigInt(1234); // "bigint"
typeof Symbol("sym"); // "symbol"
```



10 객체 래퍼 타입 피하기

Things to Remember

1

래퍼 타입이 아닌 기본형(원시형) 타입 사용하기

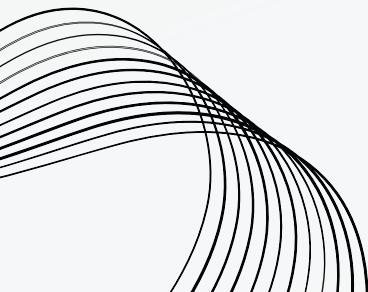
Avoid TypeScript object wrapper types. Use the primitive types instead:
string instead of String, number instead of Number, boolean instead of Boolean,
symbol instead of Symbol, and bigint instead of BigInt.

2

래퍼 객체 타입은 원시형에 대한 메서드를 사용하기 위해 사용됩니다

Symbol이나 BigInt가 아니라면 직접 사용하면 안됩니다

Understand how object wrapper types are used to provide methods on primitive values.
Avoid instantiating them or using them directly, with the exception of Symbol and BigInt.



11 잉여 속성 체크의 한계 인지하기

잉여 속성 체크

Excess Property Check

객체 리터럴

```
interface Room {
    numDoors: number;
    ceilingHeightFt: number;
}

// A. 잉여 속성 체크
const rA: Room = {
    numDoors: 1,
    ceilingHeightFt: 10,
    elephant: "present",
    // ~~~~~ Object literal may only specify known properties,
    //           and 'elephant' does not exist in type 'Room'
};

// B. 구조적 타이핑
// obj는 Room 타입의 부분집합을 포함하므로 Room에 할당 가능하며 타입 체커도 통과
const obj = {
    numDoors: 1,
    ceilingHeightFt: 10,
    elephant: "present",
};
const rB: Room = obj; // 통과됨
// 잉여 속성 체크는 할당 가능 검사 structural assignability check 와는 별도의 과정
```

임시 변수

11 잉여 속성 체크의 한계 인지하기

넓은 범위의 타입

경우에 따른 잉여 속성 체크

객체 리터럴인 경우
 객체 리터럴이 아닌 경우
 임시 변수인 경우
 타입 단언인 경우

인덱스 시그니처를 사용해
 추가적인 속성을 예상하기

```
interface Options {
  darkMode?: boolean;
  [otherOptions: string]: unknown;
}
const o: Options = { darkmode: true }; // OK
```

weak type: 선택적인 속성만 가지는 타입

타입스크립트는 값 타입과 선언 타입의
 공통된 속성이 있는지 확인하는 체크 수행

- 오타 잡기, 구조적으로 엄격하지 않음
- 선택적 필드를 포함하는 option 같은 타입에 유용

```
interface Options {
  title: string;
  darkMode?: boolean;
}

function createWindow(options: Options) {
  if (options.darkMode) {
    setDarkMode();
  }
  // ...
}

createWindow({
  title: "Spider Solitaire",
  darkmode: true,
  // ~~~~~ Object literal may only specify known properties,
  //       but 'darkmode' does not exist in type 'Options'.
  //       Did you mean to write 'darkMode'?
});

const o1: Options = document; // document는 객체 리터럴이 아니므로 - 체크 X
const o2: Options = new HTMLAnchorElement(); // 객체 리터럴이 아니므로 - 잉여속성 체크 X
const o3: Options = { darkmode: true, title: "Ski Free" }; // 객체 리터럴이므로 - 체크 O
// ~~~~~ 'darkmode' does not exist in type 'Options'...

const intermediate = { darkmode: true, title: "Ski Free" }; // right hand는 객체 리터럴이다
const o3: Options = intermediate; // right hand는 객체 리터럴이 아니다. - 체크 X

const o = { darkmode: true, title: "MS Hearts" } as Options; // 타입 단언이므로 체크 X
```

11 잉여 속성 체크의 한계 인지하기

Things to Remember

1

When you assign an object literal to a variable with a known type or pass it as an argument to a function, it undergoes excess property checking.

2

잉여 속성 체크는 오류를 찾는 효과적인 방법이지만, 구조적 할당 가능성 체크와는 역할이 다르다

검사	영어	키워드	설명
초과 속성 검사	Excess Property Checking	오타나 의도치 않은 속성을 감지	객체 리터럴을 다른 타입에 할당할 때 예상치 못한 속성이 있는지 확인하여 오류를 찾아내는 기능
구조적 할당 가능성 검사	Structural Assignability	구조적 타이핑	타입 간의 호환성을 결정할 때 사용하는 일반적인 검사

Be aware of the limits of excess property checking:
introducing an intermediate variable will remove these checks.

3

추가 속성 체크의 한계: 중간 단계의 임시 변수에 체크를 수행하지 않는다

A "weak type" is an object type with only optional properties.
For these types, assignability checks require at least one matching property.

4

weak type은 옵셔널 속성으로만 이루어진 타입이다

최소한 하나의 속성이 맞아야 한다

12 함수 표현식에 타입 적용하기

함수 전체의 타입을 정의하기

- 반복되는 함수 시그니처의 통합
- 함수 구현부 로직을 분리

```
function rollDice1(sides: number): number {} // 문장 Statement
const rollDice2 = function (sides: number): number {}; // 표현식 Expression
const rollDice3 = (sides: number): number => {}; // 표현식 expression
```

```
type DiceRollFn = (sides: number) => number;
// 함수 타입의 선언의 목적: 불필요한 코드의 반복을 줄인다 (반복되는 함수 시그니처를 통합)
// 함수 구현부가 분리되어 로직이 분명해진다.
const rollDice: DiceRollFn = (sides) => {};
```

라이브러리의 공통 함수 시그니처

공통 콜백함수를 위한 타입 선언

- JS의 [MouseEvent](#), React의 MouseEventHandler
- MouseEventHandler는 함수 전체에 적용할 수 있는 타입이다.

```
// @types/react/index.d.ts
// MouseEventHandler
// Element T를 제네릭으로 받아서 MouseEvent<T>로 EventHandler 타입에 넘김
type MouseEventHandler<T = Element> = EventHandler<MouseEvent<T>>;

// (참고용)EventHandler
type EventHandler<E extends SyntheticEvent<any>> = {
  bivarianceHack(event: E): void;
}["bivarianceHack"];
```



12 함수 표현식에 타입 적용하기

시그니처가 일치하는
다른 함수 사용하기

- fetch()

typeof fn

```
//typescript/lib/lib.dom.d.ts에 있는 fetch 타입
declare function fetch(
    input: RequestInfo,
    init?: RequestInit
): Promise<Response>;

// 다른 파일에서 활용
// 함수 전체에 타입(typeof fetch)를 적용해, 매개변수의 타입을 추론할 수 있게 한다
const checkedFetch: typeof fetch = async (input, init) => {
    const response = await fetch(input, init);
    if (!response.ok) {
        throw new Error(`Request failed: ${response.status}`);
    }
    return response;
};
```



12 함수 표현식에 타입 적용하기

Things to Remember

1

파라미터와 리턴 타입을 개별로 선언하지 않고, 전체 함수 표현식의 타입을 선언하자

Consider applying type annotations to entire function expressions,
rather than to their parameters and return type

2

반복적인 함수 시그니처를 작성하지 말고
함수 타입을 분리해내거나 & 이미 존재하는 타입을 찾도록 한다

If you're writing the same type signature repeatedly,
factor out a function type or look for an existing one.

3

라이브러리를 만들 때, 공통 콜백 타입을 제공하자

If you're a library author, provide types for common callbacks.

4

다른 함수의 시그니처를 동일하게 사용하려면 `typeof fn`을 사용한다

Use `typeof fn` to match the signature of another function,
or Parameters and a rest parameter if you need to change the return type.



THANK YOU

Reference & Link

이펙티브 타입스크립트, *Dan Vanderkam*

2장, 타입스크립트의 타입 시스템

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

TypeScript playground: <https://www.typescriptlang.org/play/?#code>

Mdn web docs: <https://developer.mozilla.org/en-US/docs/Web>

