

# 타입 공간과 값 공간의 Symbol 구분하기

# TypeScript Sybols

- Type Space / Value Space (1)

Symbol 은 타입 공간 또는 값 공간에 존재  
이름이 같더라도 속하는 공간에 서로 다른 것

```
interface Cylinder {  
  raius: number;  
  height: number;  
}  
  
const Cylinder = (radius: number, height: number) => ({ radius, height });
```

interface Cylinder 는 타입

const Cylinder 는 값

interface Cylinder ≠ const Cylinder

# TypeScript Sybols

## • Type Space / Value Space (2)

```
interface Cylinder {  
  raius: number;  
  height: number;  
}  
  
const Cylinder = (radius: number, height: number) => ({ radius, height });  
  
function calculateVolume(shape: unknown) {  
  if (shape instanceof Cylinder) {  
    shape.radius;  
    //Property 'radius' does not exist on type '{}'.  
  }  
}
```

1. 런타임 연산자 instanceof 는 값에 대해 연산
2. instanceof Cylinder 는 타입이 아니라 함수 Cylinder 를 참조
3. error 발생  
런타임 시에는 타입 관련 구문이 없다.

# Type Symbol 과 Value Symbol 의 구분 (1)

- 문맥을 통해 구분

Type	Value
type	const
interface	let
: <type>	=
as <type>	

```
type T1 = 'string literal';    // type
type T2 = 123;
const T1:T1 = 'string literal'; // value
interface Person {            // type
  first: string;
  last: string;
}

const p: Person = { first: "Jane", last: "Jacobs" }; // p: value Person: type
function email(p: Person, subject: string, body: string): Response {}
// value: email, p, subject, body
// type: Person, string , string, Response
```

## Type Symbol 과 Value Symbol 의 구분 (2)

- 컴파일 과정에서 사라진다면 해당 Symbol은 Type Symbol

```
type T1 = 'string literal';    // type
type T2 = 123;

const T1:T1 = 'string literal'; // value

interface Person {
  first: string;
  last: string;
}

const p: Person = { first: "Jane", last: "Jacobs" };

function email(p: Person, subject: string, body: string): Response {}
```

==Compile==>

```
const T1 = 'string literal'; // value
const p = { first: "Jane", last: "Jacobs" };
// p: value Person: type
function email(p, subject, body) { }
```

# typeof

```
const p: Person = { first: "Jane", last: "Jacobs" };  
function email(p: Person, subject: string, body: string): Response {}  
type T1 = typeof p;           // type T1 = Person  
type T2 = typeof email;       // type T2 = (p: Person, subject: string, body: string) => Response  
const v1 = typeof p;          // value is object  
const v2 = typeof email;      // value is function  
// const v1: "string" | "number" | "bigint" | "boolean" | "symbol" | "undefined" | "object" | "function"
```

## • 타입의 관점

값을 읽어 ts 타입을 반환

큰타입의 일부분으로 사용 / 이름을 붙이는 용도

## • 값의 관점

js 런타임의 typeof 연산자

대상 Symbol의 런타임 타입을 가리키는 문자열을 반환

ts 의 타입과 다르다.

string / number / bigint / boolean / symbol / undefined / object / function 8가지

# class 와 enum

- Type? or Value?

class 와 enum 은 상황에 따라 타입과 값 두가지 모두 가능

```
class Cylinder {  
    radius = 1;  
    height = 1;  
}  
  
function calculateVolume(shape: unknown) {  
    if (shape instanceof Cylinder) {  
        return shape.radius;  
    }  
}
```

`instance of Cylinder` 에서는 value

런타임 연산자 `instanceof` 가 오류 없이 사용 가능

클래스가 타입으로 사용될 경우 형태(속성과 메서드)가 사용

값으로 사용될 경우 생성자가 사용

# class 와 typeof

```
const v = typeof Cylinder;    //value is "function"  
type T = typeof Cylinder;    //type T = typeof Cylinder
```

js 에서 class 는 실제 함수로 구현되기 때문에 v 는 "function" 이 된다.

Cylinder 는 인스턴스의 타입 X

즉 type T = Cylinder 와 type T = typeof Cylinder 는 다르다.

```
declare let fn: T;  
const c = new fn(); // const c: Cylinder  
  
type C = InstanceType<typeof Cylinder>; // type C = Cylinder
```



## 속성 접근자

- 대괄호 표기법 `obj['field']` / ### 점표기법 `obj.field`

```
const first: Person['first'] = p['first'];
```

속성 접근자 [] 는 Type 과 Value 에서 동일

`obj['field']` 와 `obj.field` 는 값이 같아도 타입이 다를 수 있음을 명심

=> 타입 정의 시 `obj["field"]` 대괄호 표기법으로 작성

- Index의 Type

인덱스 위치에는 유니온 타입과 기본형 타입을 포함한 어떤 타입이든 사용가능

```
type PersonEl = Person['first' | 'last']; //type PersonEl = string
type Tuple = [string, number, Date];
type TupleEl = Tuple[number]; //type TupleEl = string | number | Date
```

`TupleEl` 은 Tuple 을 number 숫자 인덱스로 접근하여 그 타입을 TupleEl 의 타입으로 사용

=> `string | number | Date`

속성 접근자를 활용한 타입 정의 (item 14 에서 더 자세히)

## 그밖에 **type / value space** 에서 다른 목적으로 사용되는 경우

- 값으로 쓰이는 **this** 는 js 의 **this** 키워드(아이템 49)  
타입으로 쓰이는 **this** 는 **다형성 this** 로 ts **this** 이다. 서브클래스의 메서드 체인을 구현할 때 유용
- 값에서 **&** 와 **|** 는 AND 와 OR 비트연산 이다. 타입에서는 Intersection 과 Union 이다.
- **const** 는 새 변수를 선언하지만 **as const** 는 리터럴 또는 리터럴 표현식의 추론된 타입을 바꾼다.
- **extends** 는 서브클래스 / 서브 타입 / 제너릭 타입의 한정자를 정의 할 수 있다.
- **in** 은 루프 또는 매핑된 타입에 등장한다.

# Type 공간과 Value 공간의 혼동

ts 코드가 잘 동작하지 않는다면 타입 공간과 값 공간을 혼동해 작성했을 가능성이 높다.

## • 구조 분해 할당

```
function email(options: {person: Person, subject: string, body: string}){  
}
```

단일 객체 매개변수를 받도록 email 을 변경

js 에서는 객체 내의 각 속성을 local 변수로 만들어주는 구조 분해 할당을 사용할 수 있다.

```
function email({person: Person, subject: string, body: string}){  
  //(parameter) Person: any  
  //'Person 에 암시적으로 'any' 형식이 있다.  
  //(parameter) string: any  
  //Duplicate identifier 'string'.  
  //(parameter) string: any  
  //Duplicate identifier 'string'.  
}
```

값의 관점에서 Person 과 string 이 해석되기 때문에 오류가 발생한다.

Person 이라는 변수명과 string 이라는 이름을 가지는 두개의 변수를 생성하려한것이다.

# Type 공간과 Value 공간의 혼동

## • 구조 분해 할당

문제 해결을 위해 타입과 값을 구분

```
function email({  
  person,  
  subject,  
  body,  
}): {  
  person: Person;  
  subject: string;  
  body: string;  
}) {  
  
}
```

장황하지만 매개변수에 명명된 타입을 사용하거나 문맥에서 추론되도록 잘 동작한다.