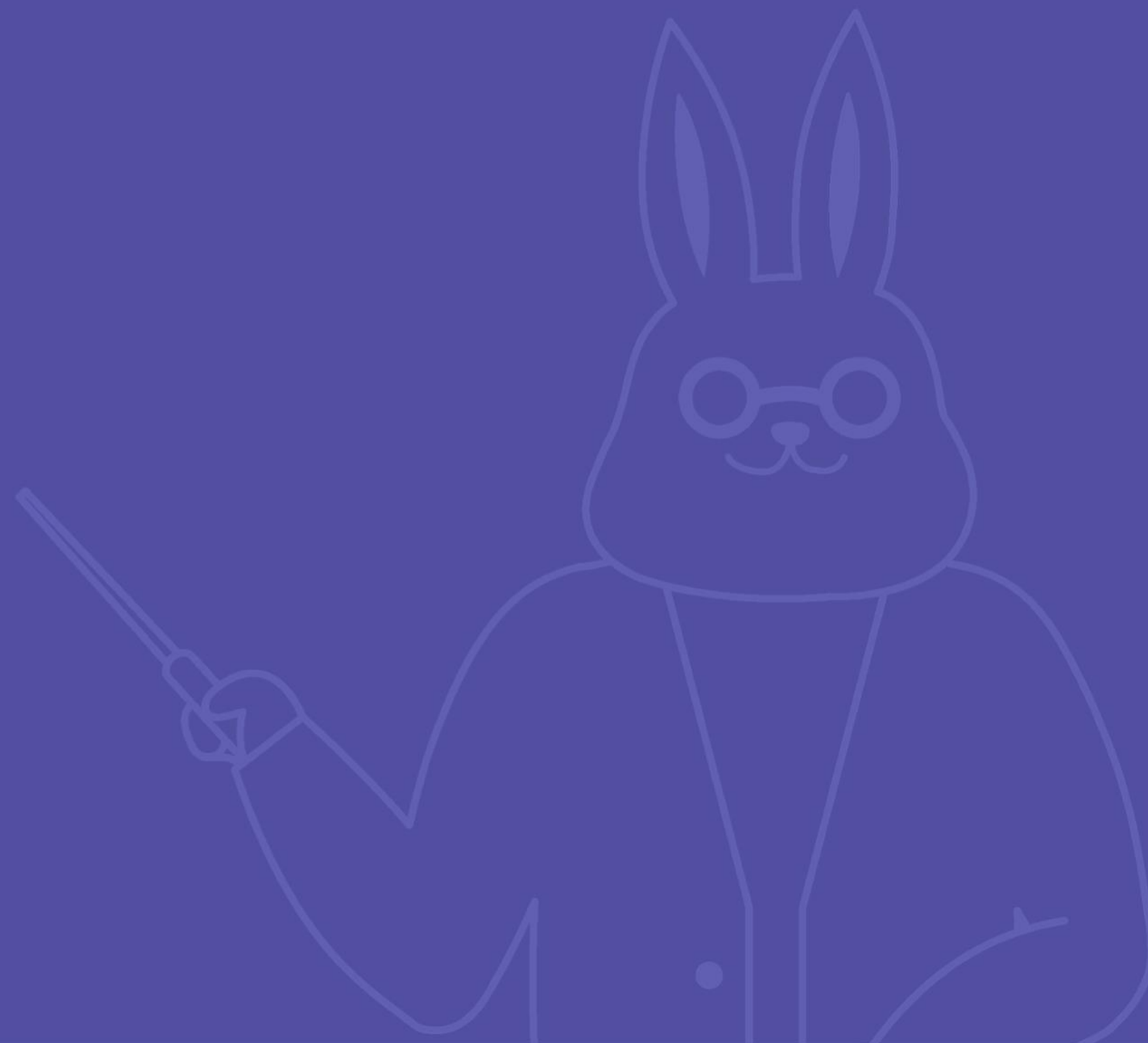




# 타입스크립트 I

00 수업 소개





# 커리큘럼



## 타입스크립트 기본

자바스크립트의 타입과 타입스크립트에서 추가 되는 타입, utility type에 대해서 살펴봅니다.



## Class

OOP의 개념, 접근제어자, get, set, static, readonly 키워드, 추상클래스에 대해서 알아봅니다. 그리고 추상클래스를 이용한 디자인 패턴 예제 코드를 살펴봅니다.

## 커리큘럼



### Interface

타입스크립트 안에서 함수와 클래스 안에서 인터페이스의 사용법에 대해서 알아봅니다. 그리고 인터페이스를 활용한 디자인 패턴 예제 코드를 살펴봅니다.



### Generic

제네릭의 개념, 제네릭을 이용한 함수, 클래스 사용법, union type, 제네릭 제약조건에 대해서 알아봅니다. 그리고 제네릭을 활용한 디자인 패턴을 예제 코드를 살펴봅니다.

## 추천대상

### 자바스크립트의 기본 문법을 알고 있는 분

자바스크립트를 어느 정도 사용해보았고, 타입의 필요성을 느끼는 분

### 타입스크립트 입문자

기본 타입부터 클래스, 인터페이스 등 타입스크립트의 기초 문법을 익히고 싶은 분

### 타입스크립트로 디자인 패턴을 적용하고 싶은 분

디자인 패턴을 이용해 객체 지향적인 코드를 작성하는 방법을 배우고 싶으신 분

# 수강목표

## 1. 타입을 활용하여 함수 클래스 코드를 작성할 수 있다.

자바스크립트에 타입을 적용하여 함수, 클래스 등을 작성하는 방법을 배웁니다.

## 2. 타입스크립트에 추가되는 문법을 알 수 있다.

타입스크립트에서 제공하는 여러가지 타입과 제네릭 같은 문법을 익힙니다.

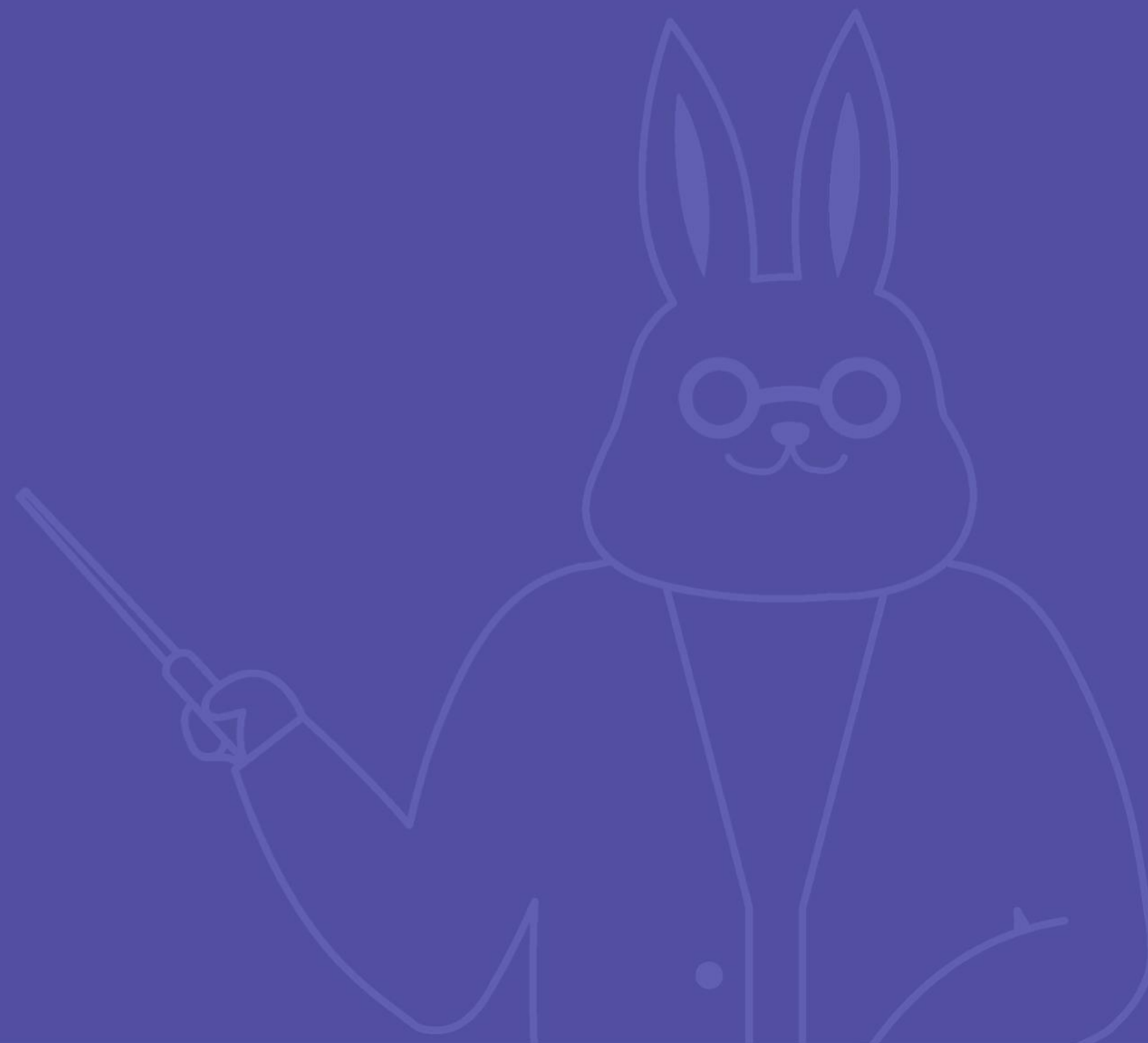
## 3. 타입스크립트에서 디자인 패턴을 적용할 수 있다.

타입스크립트를 이용한 여러 디자인 패턴 예시를 살펴봅니다.



# 타입스크립트 I

## 01 타입스크립트 기본

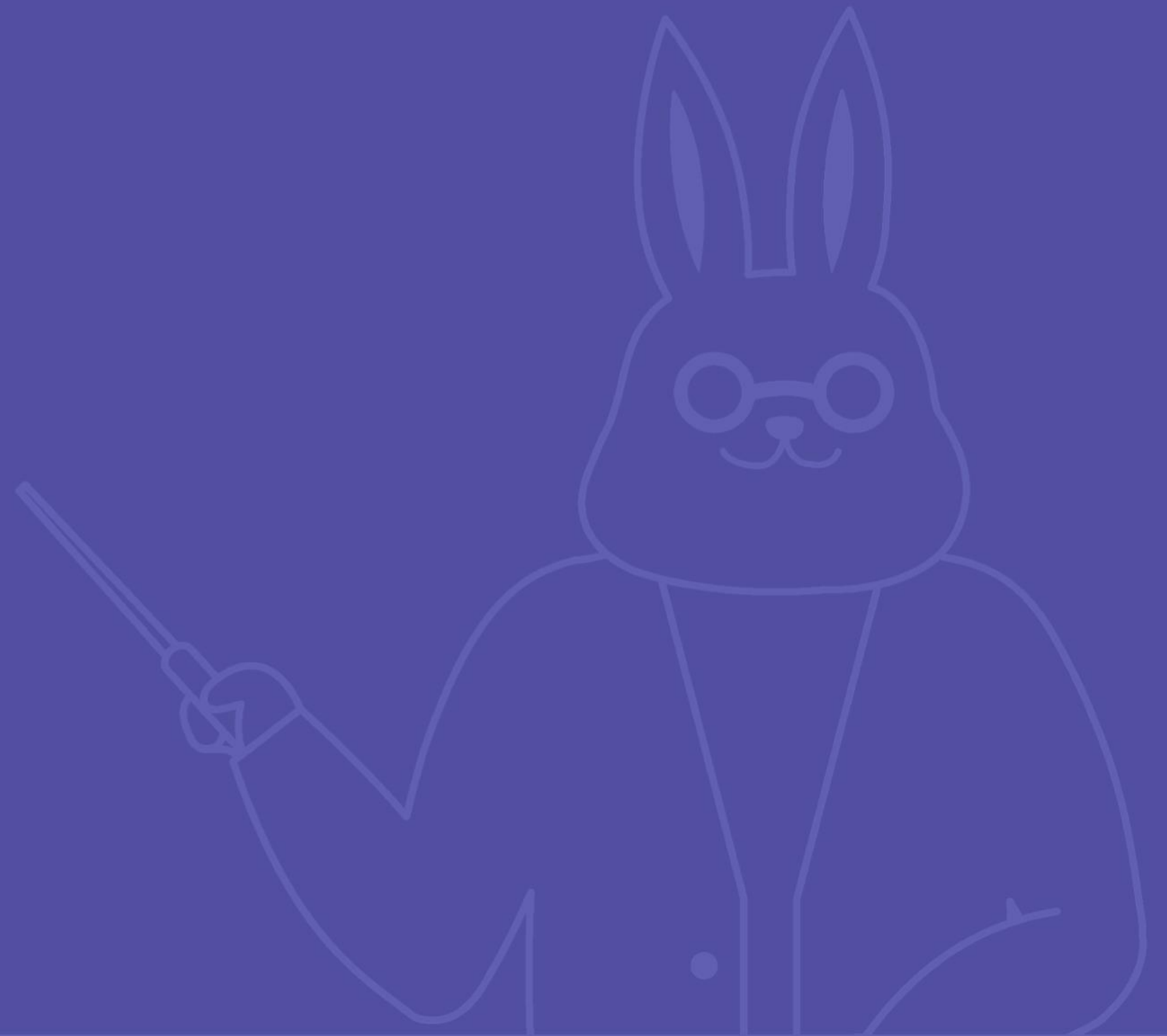


## 목차

- 01. 타입스크립트 개요
- 02. 타입스크립트의 기본 Type
- 03. Utility types
- 04. 타입스크립트를 이용해 함수 사용하기
- 05. 함수의 매개변수

01

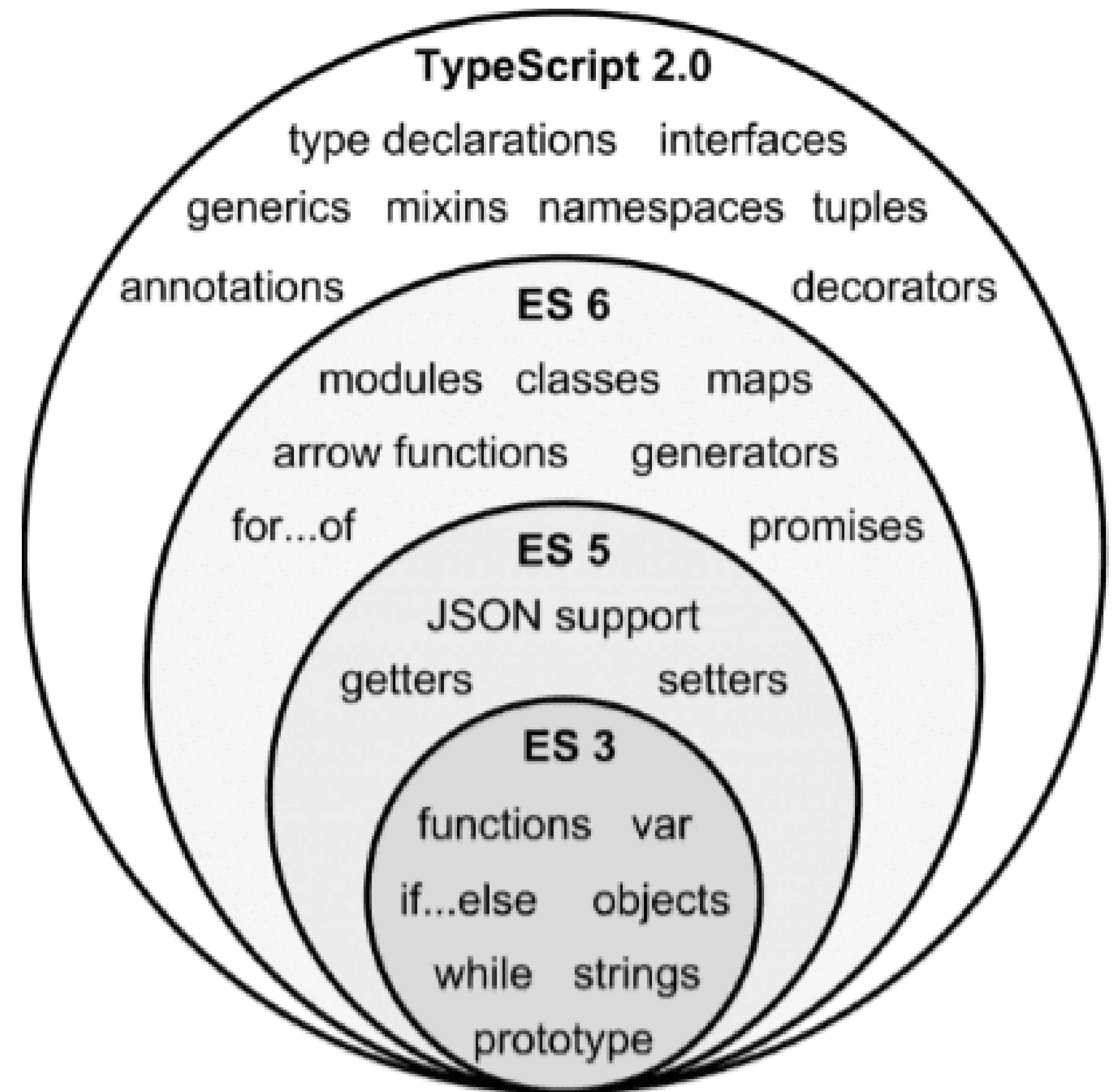
# 타입스크립트 개요



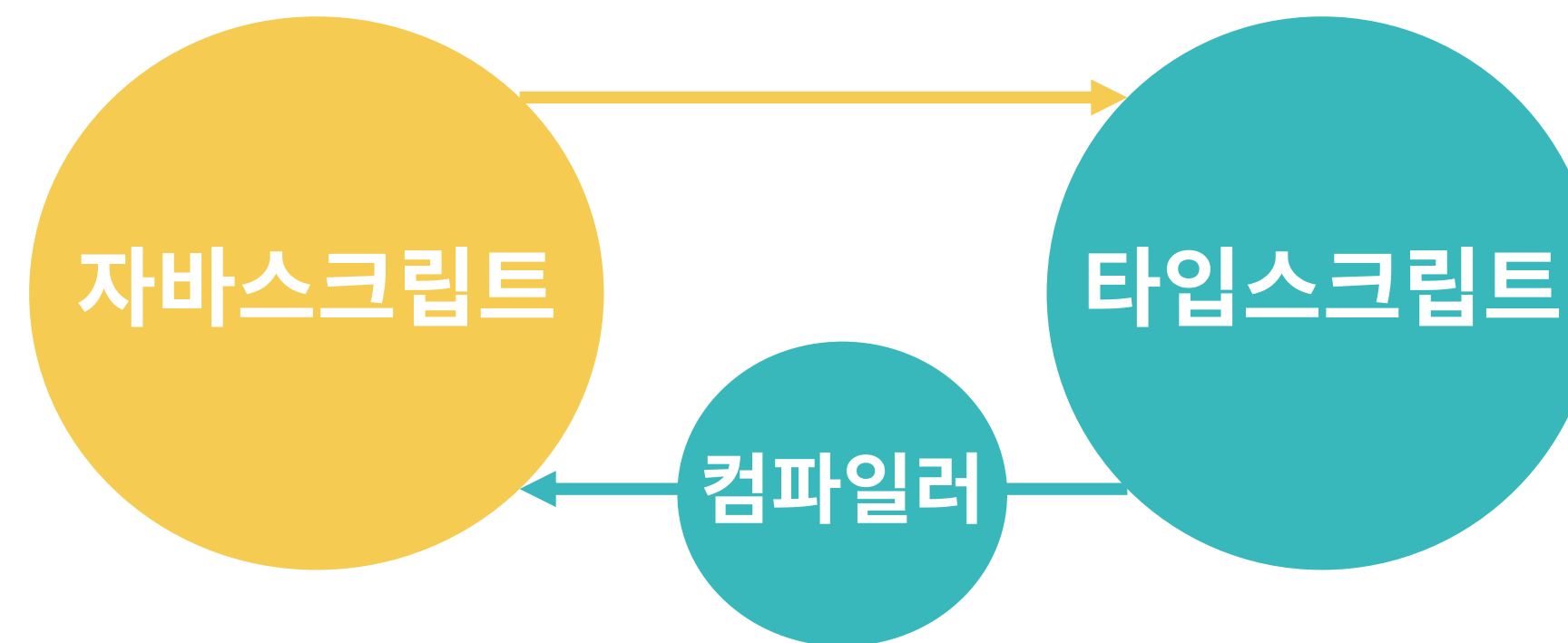


## ✔ 타입스크립트란?

- Microsoft에서 개발한 오픈 소스 언어
- 자바스크립트의 상위 집합
- 자바스크립트의 한계를 해결



## ✓ 타입스크립트란?



자바스크립트와 타입스크립트의 호환성

## ✓ 타입스크립트를 쓰는 이유

- 동적 타입을 정적으로 선언할 수 있다.
- 타입 유추를 통한 타입 제어가 가능하다.
- 컴파일 시점에 오류를 포착할 수 있다.
- JavaScript에서 찾을 수 없는 추가 코드 기능을 제공한다.

## ✓ 동적 타입을 정적으로!

### 자바스크립트

```
let a;  
  
a = 1;  
a = 'b';
```

### 타입스크립트

```
let a : number;  
  
a = 1;  
a = 'b';  
// Type 'string' is not assignable to  
type 'number'.ts(2322)
```

## ✔ 타입유추를 통한 타입 제어

### 자바스크립트

```
const sum = (a, b) => {  
  return a + b  
}
```

```
sum(1, "2") // 12
```

### 타입스크립트

```
const sum = (a: number, b: number) =>  
{  
  return a + b  
}
```

```
sum(1, 2) // 3
```

02

# 타입스크립트의 기본 Type



## ✓ TypeScript의 기본 Type

- TypeScript는 JavaScript 코드에 변수나 함수 등 Type을 정의할 수 있다.
- Type을 나타내기 위해서 타입 표기(Type Annotation)를 사용한다.
- TypeScript의 Type
  - 기본 자료형(primitive type)
  - 참조 자료형(reference type)
  - 추가 제공 자료형

## ✓ 기본 자료형

- 기본 자료형 (primitive type)

object 와 reference 형태가 아닌 실제 값을 저장하는 자료형

primitive type 내장 함수를 사용 가능한 것은 자바스크립트 처리 방식 덕분

- 종류

- string
- boolean
- number
- null
- undefined
- symbol (ES6 추가)



## ✓ 기본 자료형 (string, boolean, number)

### 코드

```
// string
// 문자열을 저장하는 타입
let str: string = "hi";

// Boolean
// 참/거짓을 저장하는 타입
let isSucceeded: boolean = true;

// number
// 부동 소수 값을 저장하는 타입
// 2진수, 8진수, 10진수, 16진수 사용 가능
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
```

## ✓ 기본 자료형 (null, undefined)

### 코드

```
// null
// 값이 의도적으로 비어 있는 상태를 저장하는 타입
let n: null = null;

// undefined
// 아무 값이 할당되지 않은 상태를 저장하는 타입
let u: undefined = undefined;

// typeof로 데이터 타입을 확인
typeof null // 'object'
typeof undefined // 'undefined'
```

```
null === undefined // false
null == undefined // true
null === null // true
null == null // true
!null // true
isNaN(1 + null) // false
isNaN(1 + undefined) // true
```

## ✓ 참조 자료형

- 참조 자료형 (reference type)

객체, 배열, 함수 등과 같은 Object형식의 타입

메모리에 값을 주소로 저장하고, 출력 시 메모리 주소와 일치하는 값을 출력

- 종류

- object
- array
- function

## ✓ 참조 자료형 (object, array)

### 코드

```
// object
// 기본 자료형에 해당하지 않는 타입
// string, boolean, number, null, undefined를 제외한
타입
function create(o: object): void{}

create({ prop: 0 }) // 성공
create([1, 2, 3]) // 성공
create("string") // error
create(false) // error
create(42) // error
create(null) // error
create(undefined) // error
```

```
// array
// 배열을 저장하는 타입
let arr: number[] = [1, 2, 3]

// 아래와 같이 제네릭을 사용한 타입 표기 가능
let arr: Array<number> = [1, 2, 3]
```

### ✓ TypeScript 제공 자료형

- 추가 제공 자료형

TypeScript에서 개발자의 편의를 위해 추가로 제공하는 타입

- 종류

- tuple
- enum
- any
- void
- never

## ✔ TypeScript 제공 자료형 (tuple, enum)

### 코드

```
// tuple
// 길이와 각 요소의 타입이 정해진 배열을 저장하는 타입
let arr: [string, number] = ["Hi", 6];

arr[1].concat("!");
// Error, 'number' does not have 'concat'

// 정의하지 않은 index 호출 시 오류
arr[3] = "hello";
// Error, Property '3' does not exist on type '[string, number]'
```

```
// enum
// 특정 값(상수)들의 집합을 저장하는 타입
enum Car { BUS, TAXI, SUV };

let bus: Car = Car.BUS;
let bus: Car = Car[0]; // 인덱스 번호로 접근

// 인덱스를 사용자 편의로 변경
enum Car { BUS = 1, TAXI = 2, SUV = 3 };
let taxi: String = Car[2];

enum Car { BUS = 2, TAXI, SUV };
let taxi: String = Car[3];
```

## ✔ TypeScript 제공 자료형 (any, void)

### 코드

```
// any
// 모든 타입을 저장 가능
// 컴파일 중 타입 검사를 하지 않음
let str: any = "hi";
let num: any = 10;
let arr: any = ["a", 2, true];

// void
// 보통 함수에서 반환 값이 없을 때, any의 반대 타입
// 변수에는 undefined와 null만 할당하고, 함수에는 반환 값을 설정할 수 없는 타입
let unknown: void = undefined;

function sayHi(): void {
  console.log("hi");
}
```

## ✔ TypeScript 제공 자료형 (never)

### 코드

```
// never
// 발생할 수 없는 타입
// 항상 오류를 발생시키거나 절대 반환하지 않는 반환 타입
// 종료되지 않는 함수
function neverEnd(): never {
  while (true) {}
}
```

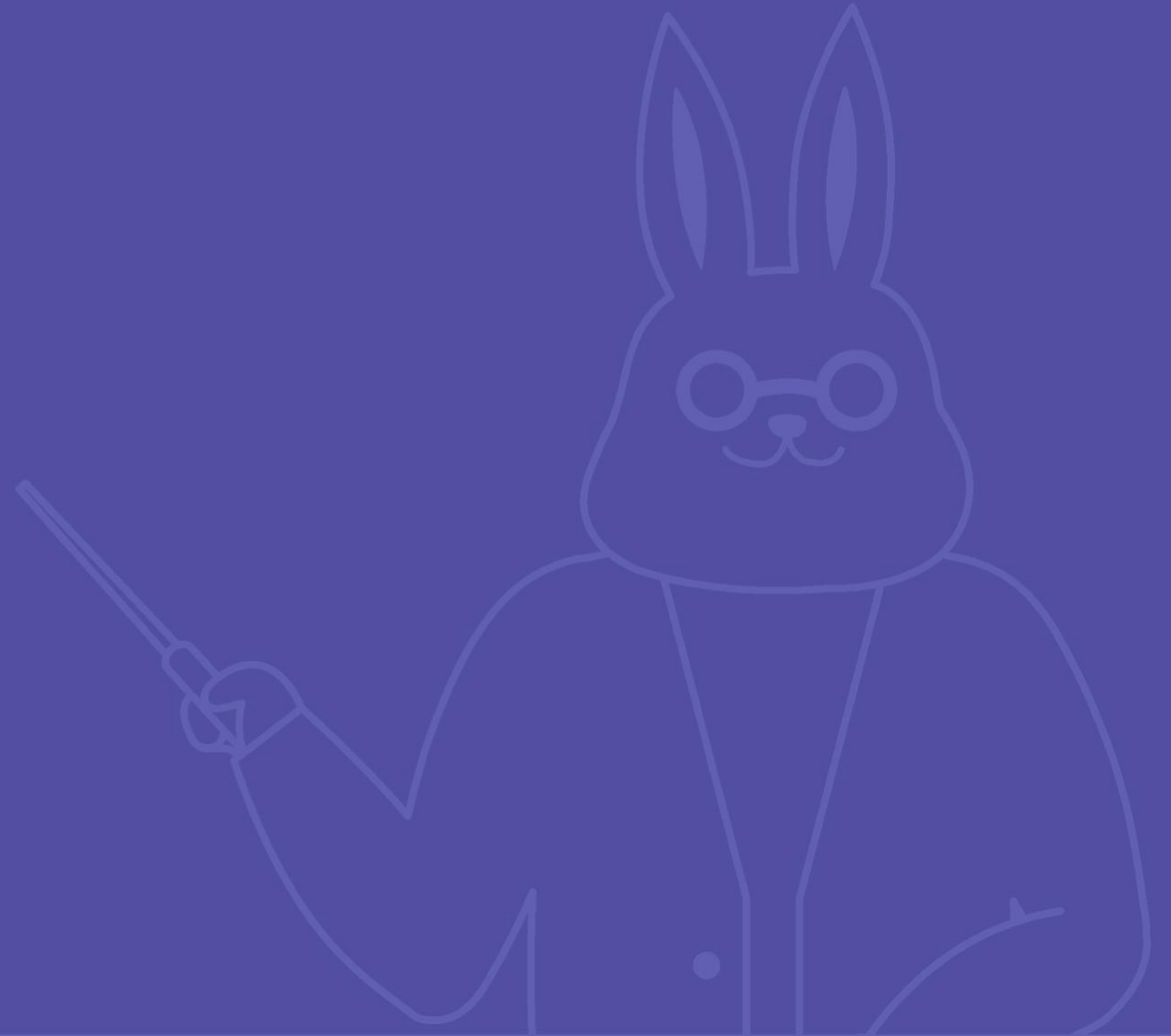
```
// Error: A function returning 'never' cannot
// have a reachable end point.ts(2534)
function neverEnd(): never {
  while (true) {
    break;
  }
}

// 항상 오류를 발생시키는 함수
function error(message: string): never {
  throw new Error(message);
}
```



03

# Utility types



## ✓ Utility types

- TypeScript는 공통 타입 변환을 용이하게 하기 위해 유틸리티 타입을 제공한다.
- 유틸리티 타입은 전역으로 사용 가능하다.
- 종류
  - Partial<T>, Readonly<T>
  - Record<T>, Pick<T,K>
  - Omit<T,K>, Exclude<T,U>, Extract<T,U>
  - NonNullable<T>, Parameters<T>, ConstructorParameters<T>
  - ReturnType<T>, Required<T>

## ✓ Partial<T>, Readonly<T>

### Partial<T>

```
interface Todo {
  title: string;
  description: string;
}

function updateTodo(todo: Todo, fieldsToUpdate: Partial<Todo>) {
  return { ...todo, ...fieldsToUpdate };
}

const todo1 = {
  title: 'organize desk',
  description: 'clear clutter',
};

const todo2 = updateTodo(todo1, {
  description: 'throw out trash',
});
```

- 프로퍼티를 선택적으로 만드는 타입을 구성한다.
- 주어진 타입의 모든 하위 타입 집합을 나타내는 타입을 반환한다.

### Readonly<T>

```
interface Todo {
  title: string;
}

const todo: Readonly<Todo> = {
  title: 'Delete inactive users',
};

// 사용 예: frozen 객체의 프로퍼티에 재할당 방지
todo.title = 'Hello'; // Error: Cannot assign to 'title' because it is a read-only property
```

프로퍼티를 읽기 전용(readonly)으로 설정한 타입을 구성한다.

## ✓ Record<T, K>, Pick<T,K>

### Record<T, K>

```
interface PageInfo {
  title: string;
}

type Page = 'home' | 'about' | 'contact';

const x: Record<Page, PageInfo> = {
  about: { title: 'about' },
  contact: { title: 'contact' },
  home: { subTitile: 'home' },
  // Error: '{ subTitile: string; }' is not assignable
  main: { title: 'home' },
  // Error: main is not assignable to type 'Page'.
};
```

프로퍼티의 집합 K로 타입을 구성한다.  
타입의 프로퍼티들을 다른 타입에 매핑시키는 데  
사용한다.

### Pick<T, K>

```
interface Todo {
  title: string;
  description: string;
  completed: boolean;
}

type TodoPreview = Pick<Todo, 'title' | 'completed'>;

const todo: TodoPreview = {
  title: 'Clean room',
  completed: false,
  description: 'description'
  // Error: 'description' is not assignable to type
};
```

프로퍼티 K의 집합을 선택해 타입을  
구성한다.

## ✓ Omit<T,K>

Omit<T, K>

```
interface Todo {
  title: string;
  description: string;
  completed: boolean;
}

type TodoPreview = Omit<Todo, 'description'>;

const todo: TodoPreview = {
  title: 'Clean room',
  completed: false,
  description: 'description' // Error: 'description' is not assignable to type
};
```

모든 프로퍼티를 선택한 다음 K를 제거한 타입을 구성한다.

✓ **Exclude<T,U>, Extract<T,U>**

## Exclude&lt;T, U&gt;

```
type T0 = Exclude<"a" | "b" | "c", "a">;  
// "b" | "c"  
type T1 = Exclude<"a" | "b" | "c", "a" | "b">;  
// "c"  
type T2 = Exclude<string | number | (() => void)  
, Function>; // string | number
```

T에서 U에 할당할 수 있는 모든 속성을 제외한 타입을 구성한다.

## Extract&lt;T, U&gt;

```
type T0 = Extract<"a" | "b" | "c", "a" | "f">;  
// "a"  
type T1 = Extract<string | number | (() => void)  
, Function>; // (() => void
```

T에서 U에 할당할 수 있는 모든 속성을 추출하여 타입을 구성한다.

## ✓ NonNullable<T>, Parameters<T>

### NonNullable<T>

```
type T0 = NonNullable<string | number | undefined>; // string | number
type T1 = NonNullable<string[] | null | undefined>; // string[]
```

null 과 undefined를 제외한 타입이다.

### Parameters<T>

```
declare function f1(arg: { a: number, b: string }): void
type T0 = Parameters<() => string>; // []
type T1 = Parameters<(s: string) => void>; // [string]
type T2 = Parameters<(<T>(arg: T) => T)>; // [unknown]
type T4 = Parameters<typeof f1>; // [{ a: number, b: string }]
type T5 = Parameters<any>; // unknown[]
type T6 = Parameters<never>; // never
type T7 = Parameters<string>; // 오류
type T8 = Parameters<Function>; // 오류
```

함수 타입 T의 매개변수 타입들의 튜플 타입을 구성한다.

## ✓ ConstructorParameters<T>

ConstructorParameters<T>

```
type T10 = ConstructorParameters<ErrorConstructor>; // [(string | undefined)?]
type T1 = ConstructorParameters<FunctionConstructor>; // string[]
type T2 = ConstructorParameters<RegExpConstructor>; // [string, (string | undefined)?]

interface I1 {
  new(args: string): Function;
}
type T12 = ConstructorParameters<I1>; // [string]

function f1(a: T12) {
  a[0]
  a[1] // Error: Tuple type '[args: string]' of length '1' has no element at index '1'.
}
```

생성자 함수 타입의 모든 매개변수 타입을 추출한다.

모든 매개변수 타입을 가지는 튜플 타입(T가 함수가 아닌 경우 never)을 생성한다.



## ✓ ReturnType<T>, Required<T>

### ReturnType<T>

```
declare function f1(): { a: number, b: string }
type T0 = ReturnType<() => string>; // string
type T1 = ReturnType<(s: string) => void>; // void
type T2 = ReturnType<(<T>() => T)>; // {}
type T3 = ReturnType<(<T extends U, U extends number[]>() => T)>; // number[]
type T4 = ReturnType<typeof f1>; // { a: number, b: string }
type T5 = ReturnType<any>; // any
type T6 = ReturnType<never>; // any
type T7 = ReturnType<string>; // 오류
type T8 = ReturnType<Function>; // 오류
```

함수 T의 반환 타입으로 구성된 타입을 생성한다.

### Required<T>

```
interface Props {
  a?: number;
  b?: string;
};

const obj: Props = { a: 5 };

const obj2: Required<Props> = { a: 5 };
// Error: Property 'b' is missing in type '{ a: number; }'
```

T의 모든 프로퍼티가 필수로 설정된 타입을 구성한다.

04

# TypeScript를 이용해 함수 사용하기



## ✓ 용어 정리

## 코드

```
function add(x, y) {  
  return x + y;  
}
```

```
add(2, 5);
```

- 함수를 정의할 때 사용되는 변수를 매개 변수라고 한다.
- 함수를 호출할 때 사용되는 값을 인수라고 한다.
- 인자 값 == 매개변수 == Parameter

## ✓ 일급 객체

- 일급 객체(first-class object)

다른 객체들에 일반적으로 적용 가능한 연산을 모두 지원하는 객체를 일급 객체라고 한다.

- 일급 객체의 조건

다른 함수에 매개변수로 제공할 수 있다.

함수에서 반환 가능하다.

변수에 할당 가능하다.

- JavaScript와 TypeScript의 함수는 일급 객체(first-class object)이다.

## ✓ 함수 선언 방법 (5가지)

### 함수 선언식

```
function world(name) {  
  return 'hello ${name}';  
}
```

### 함수 표현식

```
let world2 = function (name) {  
  return 'hello ${name}';  
}
```

## ✓ 함수 선언 방법 (5가지)

### 화살표 함수 표현식

```
let world3 = (name) => {  
  return 'hello ${name}';  
}
```

### 단축형 화살표 함수 표현식

```
let world4 = (name) => 'hello ${name}';
```

## ✓ 함수 선언 방법 (5가지)

### 함수 생성자

```
let world5 = new Function("name",  
  'return "hello " + name');
```

함수 생성자는 되도록 사용을 권장하지 않는다.

## ✓ TypeScript를 이용해 함수 사용하기

- TypeScript 함수 작성 시 반환 타입을 추론 하도록 하는 걸 권장한다.
- 함수의 매개 변수와 인수의 타입이 호환 가능하게 작성한다.
- 인수의 타입을 잘못 전달하면 에러가 발생한다.



## ✔ TypeScript를 이용해 함수 사용하기

### 코드

```
// 함수 선언식
function world(name: string): string {
  return 'hello ${name}';
}
```

```
// 함수 표현식
let world2 = function (name: string): string {
  return 'hello ${name}';
}
```

```
// 화살표 함수 표현식
let world3 = (name: string): string => {
  return 'hello ${name}';
}
```

```
// 단축형 화살표 함수 표현식
let world4 = (name: string): string => 'hello ${name}';
```

## ✔ 타입 추론

- TypeScript 컴파일러는 방정식의 한쪽에만 타입이 있더라도 타입을 추론할 수 있다.
- 이러한 타입 추론 형태를 “*contextual typing*”이라고 한다.
- 이를 통해 프로그램에서 타입을 유지하기 위한 노력을 줄일 수 있다.

### 일반적인 함수

```
let f12 = function (x: number, y: number): number {  
  return x + y;  
}
```

### contextual typing

```
// 매개변수 x와 y는 number 타입  
let f12: (baseValue: number, increment: number) => number = function (x, y) {  
  return x + y;  
}
```

05

# 함수의 매개변수



## ✓ 기본 매개변수 (Parameter)

- 함수에 주어진 인자의 수는 함수가 기대하는 매개변수의 수와 일치해야 한다.

### 코드

```
function buildName(firstName: string, lastName: string) {  
    return firstName + " " + lastName;  
}  
  
let result1 = buildName("Bob"); // Error: Expected 2 arguments, but got 1  
let result2 = buildName("Bob", "Adams", "Sr."); // Error: Expected 2 arguments, but got 3  
let result3 = buildName("Bob", "Adams");
```

## ✓ 선택적 매개변수 (Optional Parameter)

- JavaScript에서는 모든 매개변수가 선택적으로, 인수가 없다면 undefined가 된다.
- TypeScript에서도 선택적 매개변수를 사용할 수 있다. (변수명 뒤에 '?')

### 코드

```
function buildName(firstName: string, lastName?: string) {  
  if (lastName) return firstName + " " + lastName;  
  else return firstName;  
}  
  
let result1 = buildName("Bob");  
let result2 = buildName("Bob", "Adams");  
let result3 = buildName("Bob", "Adams", "Sr."); // Error: Expected 2 arguments, but got 3
```

## ✓ 기본-초기화 매개변수 (Default Parameter)

- TypeScript에서는 값을 제공하지 않거나, undefined로 했을 때에 매개변수의 값 할당 가능

### 코드

```
function buildName(firstName: string, lastName = "Smith") {  
    return firstName + " " + lastName;  
}  
  
let result1 = buildName("Bob"); // "Bob Smith"  
let result2 = buildName("Bob", undefined); // "Bob Smith"  
let result3 = buildName("Bob", "Adams"); // "Bob Adams"  
let result4 = buildName("Bob", "Adams", "Sr."); // Error: Expected 1-2 arguments, but got 3.
```

## ✓ 나머지 매개변수 (Rest Parameters)

- 컴파일러는 생략 부호(...) 뒤의 인자 배열을 빌드해 함수에서 사용할 수 있다.
- 나머지 매개변수는 매개변수의 수를 무한으로 취급한다.
- 아무것도 넘겨주지 않을 수도 있다.

### 코드

```
function buildName1(firstName: string, ...restOfName: string[]) {  
  // restOfName = [ 'Samuel', 'Lucas', 'MacKinzie' ]  
  return firstName + " " + restOfName.join(" ");  
}  
  
let employeeName = buildName1("Joseph", "Samuel", "Lucas", "MacKinzie"); // "Joseph Samuel Lucas  
MacKinzie"
```

# 크레딧

/\* elice \*/

코스 매니저

이재성

콘텐츠 제작자

김준영

강사

김준영

감수자

이재성

디자이너

김루미



# 연락처

TEL

070-4633-2015

WEB

<https://elice.io>

E-MAIL

[contact@elice.io](mailto:contact@elice.io)

