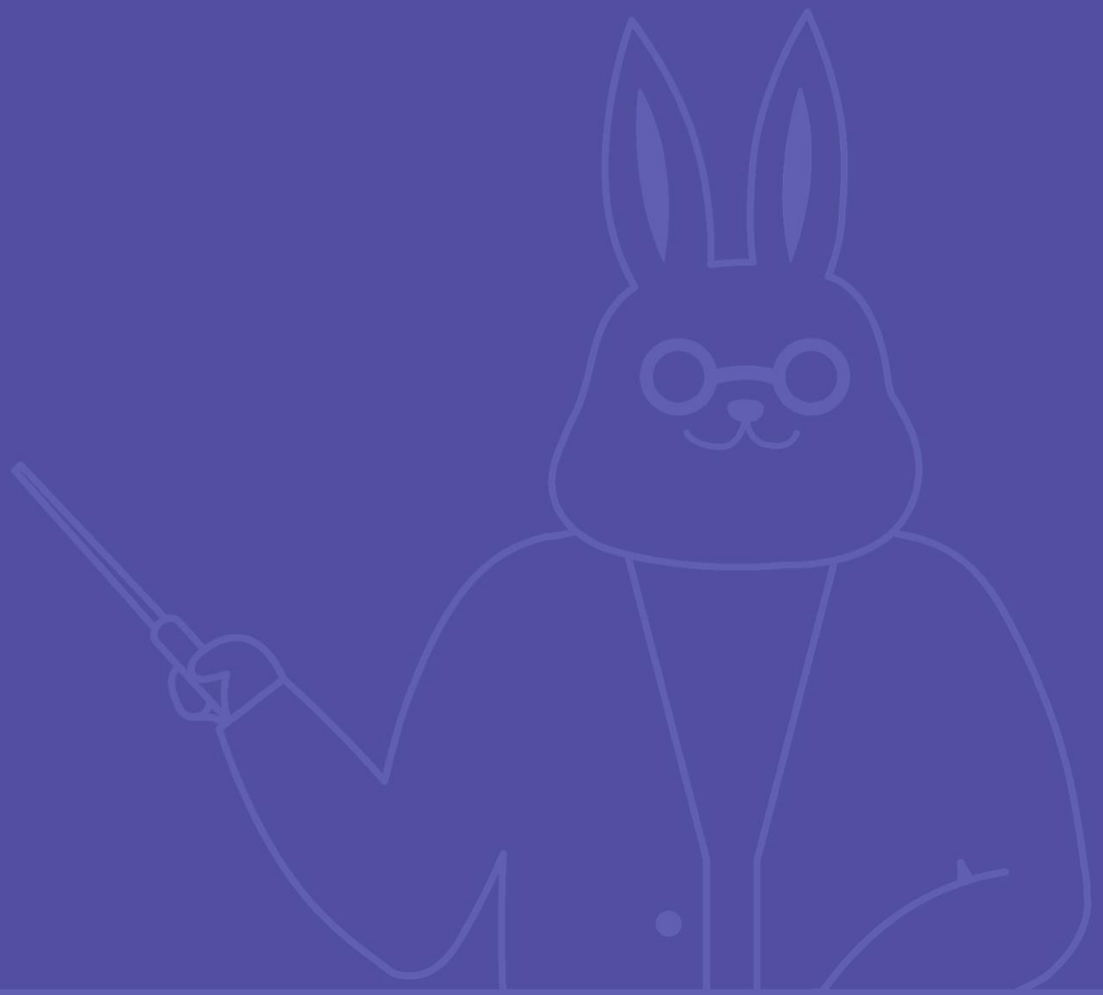
타입스크립트 II

02 데코레이터



목차

🎯 수강목표

1. 데코레이터의 등장 배경과 정의 이해하기

일급 객체, 고차 함수, 클로저 등 데코레이터를 쓰기 전 알아두어야 할 자바스크립트 개념들을 먼저 익히고, 이 개념들이 데코레이터에 어떻게 적용되고 데코레이터의 등장에 어떤 영향을 주었는지 알아봅니다.

2. 데코레이터의 5가지 타입과 타입별 용도 익히기

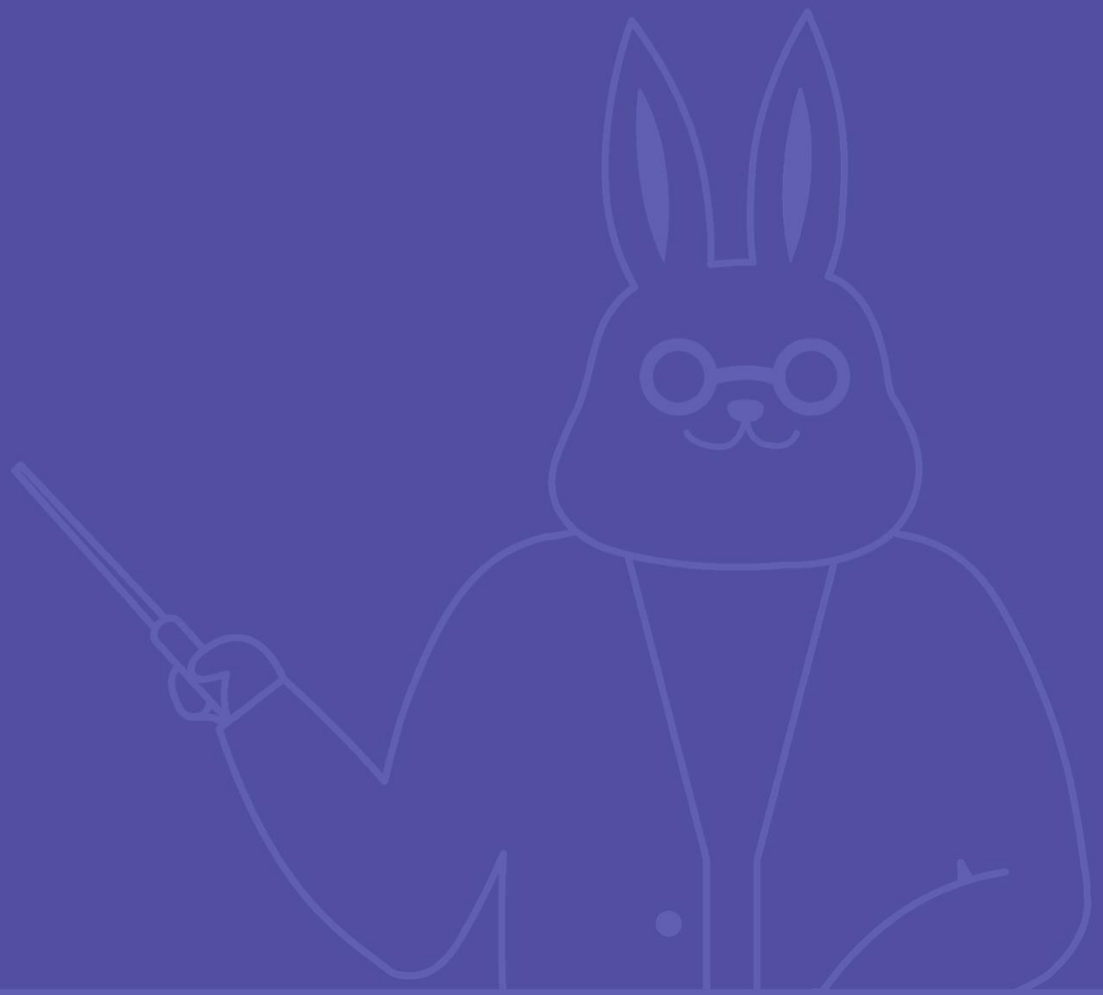
타입스크립트 데코레이터에는 class, method, property, accessor, parameter 5가지 유형의 데코레이터가 있습니다. 유형별로 데코레이터를 정의하는 방법과 실무에서 쓰일 법한 코드 위주로 유형별 용도를 익힙니다.

3. 데코레이터의 동작 원리 이해하기

컴파일된 자바스크립트 코드를 통해서 데코레이터의 평가와 적용 순서, 데코레이터 합성에 대한 원리를 이해할 수 있습니다. 자바스크립트에서 데코레이터 유형을 구별하는 방식, 선언한 순서와 반대로 호출되는 이유를 코드를 통해 배울 수 있습니다.

01

데코레이터의 정의



데코레이터의 필요성

데코레이터를 쓰지 않은 코드

```ts
type Post = {
  title?: string;
  content?: string;
  author?: string;
};
declare interface DbModel<T> {
  getOne: (id: string) => T;
  create: (input: T) => T;
  update: (model: T, input: T) => T;
}
```

```ts
class PostService {
  constructor(private db: DbModel<Post>) {}

  upsertPost(id: string, input: Post): Post {
    console.time("upsertPost");
    const post = this.db.getOne(id);
    if (post) {
      const updatedPost = this.db.update(post, { ...post, ...input });
      console.timeEnd("upsertPost");
      return updatedPost;
    }
    const newPost = this.db.create(input);
    console.timeEnd("upsertPost");
    return newPost;
  }

  getPost(id: string) {
    console.time("getPost");
    const post = this.db.getOne(id);
    console.timeEnd("    getPost");
    return post;
  }
}
```

데코레이터의 필요성

데코레이터를 쓰지 않은 코드의 불편한 점

1. 가독성
2. 반복되는 코드
3. 단일책임 원칙에 어긋남

*단일책임원칙(SRP, Single Responsibility Principle):
객체지향 5대 원칙인 SOLID 원칙 중 하나로, 프로그램에서 모든 모듈, 클래스, 함수는 한 번에 하나의 역할만 책임져야 한다는 원칙. 즉, 한 번에 한 가지 일만 해야 한다.

```
class PostService {
 constructor(private db: DbModel<Post>) {}

 upsertPost(id: string, input: Post): Post {
  console.time("upsertPost");
  const post = this.db.getOne(id);
  if (post) {
   const updatedPost = this.db.update(post, { ...post, ...input });
   console.timeEnd("upsertPost");
   return updatedPost;
  }
  const newPost = this.db.create(input);
  console.timeEnd("upsertPost");
  return newPost;
 }

 getPost(id: string) {
  console.time("getPost");
  const post = this.db.getOne(id);
  console.timeEnd("   getPost");
  return post;
 }
}
```

함수가 한 가지 일만 하면서 가독성을 높이고 반복되는 코드를 줄이려면? 데코레이터!

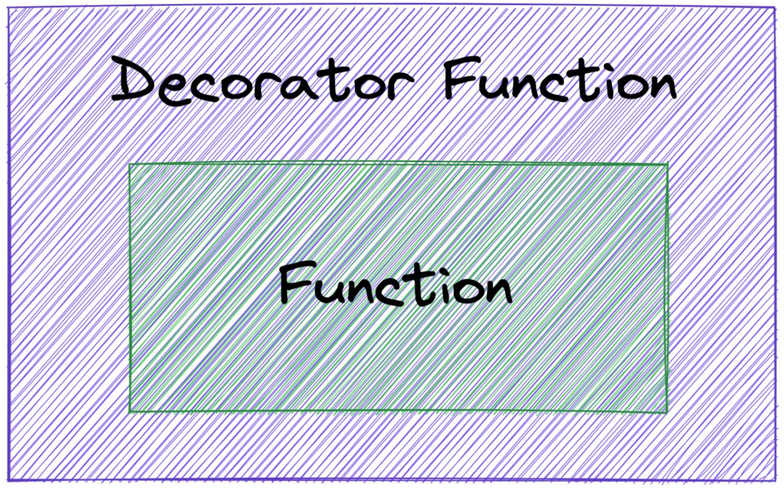
데코레이터 사용 전

```
class PostService {
 constructor(private db: DbModel<Post>) {}

 upsertPost(id: string, input: Post): Post {
  console.time("upsertPost");
  const post = this.db.getOne(id);
  if (post) {
   const updatedPost = this.db.update(post,
{ ...post, ...input });
   console.timeEnd("upsertPost");
   return updatedPost;
  }
  const newPost = this.db.create(input);
  console.timeEnd("upsertPost");
  return newPost;
 }

 getPost(id: string) {
  console.time("getPost");
  const post = this.db.getOne(id);
  console.timeEnd("getPost");
  return post;
 }
}
```

데코레이터 사용 후

```
class PostService {
 constructor(private db: DbModel<Post>) {}

 @timer("upsertPost")
 upsertPost(id: string, input: Post): Post {
  const post = this.db.getOne(id);
  if (post) {
   return this.db.update(post,
{ ...post, ...input });
  }
  return this.db.create(input);
 }

 @timer("getPost")
 getPost(id: string) {
  return this.db.getOne(id);
 }
}
```

데코레이터, Decorator



함수

함수를 감싸는 함수

GoF에 소개된 구조적 디자인 패턴 중 하나로서,
기존 함수를 바꾸지 않고 함수를 관찰, 수정, 재정의(오버라이딩)할 수 있는 함수

데코레이터 사용 시 주의할 점: 표준화되지 않은 기능

> 데코레이터는 처음에 앵귤러 프레임워크를 지원하기 위해 추가되었습니다. 아직까지도 표준화되지 않은 기능이므로 사용중인 데코레이터 비표준화되거나 호환성이 깨질 가능성이 있습니다. 타입스크립트 코드에서 모든 타입 정보를 제거하면 자바스크립트가 되지만 enum, 매개변수 속성, 트리플 슬래시 입포트, 데코레이터는 타입 정보를 제거한다고 자바스크립트가 되지 않습니다. 타입스크립트의 역할을 명확하게 하려면 데코레이터는 사용하지 않는 게 좋습니다.

인용 출처: 책 Effective Typescript p.267, 댄 밴더캄

tc39에서 제안된 stage2의 기능 (tc39 process에 의하면 실험적 기능에 속함)

때문에 자바스크립트에서는 babel 플러그인을 사용해서 데코레이터 기능을 사용.

타입스크립트에서도 tsconfig.json에 experimentalDecorator 옵션을 true로 설정해줘야 함.

하지만 Nestjs나 Angular, TypeORM 등 여러 오픈소스에서

이미 널리 쓰이고 있기 때문에 쉽게 사라지진 않을 것 같다는 의견이 많음.

02

데코레이터를 쓰기 전 알아야 할 자바스크립트 개념

Decorator 코드에 담긴 자바스크립트 개념들

timer Decorator

```
function timer(label: string) {
  return function (
    target: any,
    propertyKey: string,
    descriptor: PropertyDescriptor
  ) {
    const pureFn = descriptor.value;

    descriptor.value = function (...args: any[]) {
      console.time(label);

      const result = pureFn.apply(this, args);

      console.timeEnd(label);
      return result;
    };
  };
}
```

함수를 반환하는 함수: 고차 함수

함수의 메타데이터: 설명자

외부 함수의 매개변수에 접근: 클로저

함수의 변수 할당: 일급 객체

자바스크립트의 Function은 '일급 객체(First Class Object)'

> First and second class objects. In ALGOL a real number may appear in an expression or be assigned to a variable, and either may appear as an actual parameter in a procedure call. A procedure, on the other hand, may only appear in another procedure call either as the operator (the most common case) or as one of the actual parameters.

*ALGOL: Algorithmic Language의 준말

인용 출처: "Fundamental Concepts in Programming Languages" p.32, Christopher Strachey

"일급 객체는 ALGOL에서는 값이 표현식에 나타나거나 변수에 할당될 수 있다. 그리고 함수 호출 시 하나의 매개변수로 나타날 수도 있다.
또 어떤 함수 호출에서는 (흔한 경우) 연산자나 매개변수 중 하나로서 나타날 수 있다."

First Class Citizen, First Class Object는 **객체로써 사용되는 데 제약이 없는 상태**.

즉, **일반적으로 객체가 사용될 수 있는 곳에는 함수도 쓰일 수 있다**.

‘일급 객체(First Class Object)’의 특징

```
function timer(label: string, callback: () => void) {
 return function (
  target: any,
  propertyKey: string,
  descriptor: PropertyDescriptor
 ) {
  const pureFn = descriptor.value;

  descriptor.value = function (...args: any[]) {
   console.time(label);

   const result = pureFn.apply(this, args);

   callback();

   console.timeEnd(label);
   return result;
  };

  console.log(pureFn === descriptor.value);
 };
}
```

모든 일급 객체는 함수의 실질적인 매개변수가 될 수 있다.

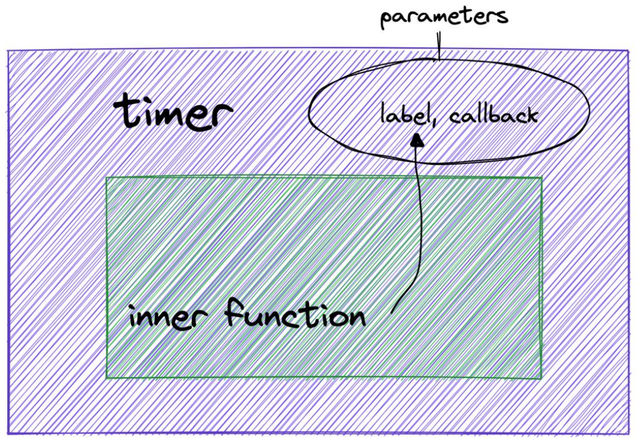
모든 일급 객체는 함수의 반환값이 될 수 있다.

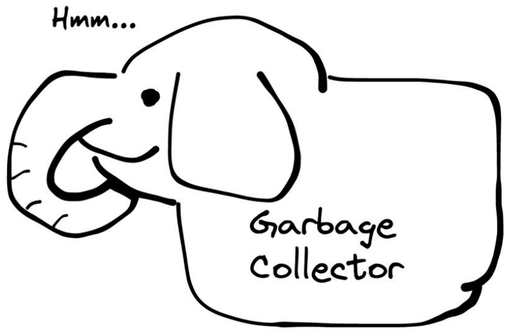
모든 일급 객체는 할당의 대상이 될 수 있다.

모든 일급 객체는 비교 연산(==, equal)을 적용할 수 있다.

자바스크립트의 클로저(Closure)

외부함수의 실행이 끝나도 내부함수에서 외부함수의 context에 접근할 수 있는 것





timer 함수가 호출되었을 때 함수의 context 생명주기는 끝나고
context의 정보는 **가비지 컬렉터(쓰레기 수집가)에 의해 메모리에서 삭제**되어야 함

그런데 timer의 내부 함수는 어떻게 timer의 label 파라미터에 접근할 수 있는 걸까?

자바스크립트의 클로저(Closure)

예시

```
function timer(label: string) {
  return function (
    target: any,
    propertyKey: string,
    descriptor: PropertyDescriptor
  ) {
    const pureFn = descriptor.value;

    descriptor.value = function (...args: any[]) {
      console.time(label);

      const result = pureFn.apply(this, args);

      console.timeEnd(label);
      return result;
    };
  };
}
```

가비지 컬렉터는 **값을 참조하는 곳이 한 곳이라도 있다면 그 값을 제거하지 않음**

내부함수가 timer 호출 이후에도 timer의 파라미터에 접근할 수 있는 이유는, **timer 함수가 반환한 내부 함수에서 label 파라미터를 참조하고 있기 때문**

자바스크립트의 고차 함수(Higher-Order Function)

예시

```
function higherOrderFn(message:
string, callback: () => void) {
  return function () {
    console.log(message);
    callback();
  };
}
```

함수를 반환하는 함수

함수를 파라미터로 받아 반환하기도 한다.

고차 함수(Higher-Order Function)의 용도

- 커링: 함수의 매개변수를 받는 시점을 늦출 때
- 함수의 실행 시점을 미룰 때(ex. jest에서 expect - throw, react - onClick)
- 팩토리 함수(인스턴스 객체나 함수를 생성하여 반환하는 함수) 만들 때
- 훅, 데코레이터를 만들 때
- 반복되는 코드 줄일 때

데코레이터와 고차 함수의 관계

예시

```
function log(instance, fn) {
  return function () {
    console.log('start - ${fn.name}');
    fn.call(instance);
    console.log('end - ${fn.name}');
  };
}

class Human {
  constructor(public name: string) {}

  hello() {
    return "hello" + this.name;
  }
}

const elice = new Human();
const helloWithLog = log(elice, elice.hello);
helloWithLog();
```

클래스 메서드는 고차 함수의 파라미터로서 전달했을 때 메서드(fn)는 인스턴스에 대한 this 바인딩을 잃어버린다.

따라서 higher order 함수를 호출할 때 인스턴스와 메서드를 함께 인자로 전달해줘야 함.

이러한 불편함, 그리고 다른 언어에서의 데코레이터 문법에 익숙한 개발자들을 위해 자바스크립트에도 데코레이터 문법이 제안됨

한편, 데코레이터 팩토리를 이해하기 위해서도 고차 함수를 알아야 한다.

Property Descriptor

자바스크립트 객체는

객체의 프로퍼티의 속성, 메타데이터를 갖고 있는

Property Descriptor를 갖는다.

Property Descriptor는 그 자체로 객체이다.

그럼 Property Descriptor는 어떤 속성을 가지고 있을까?

Property Descriptor 객체의 속성

예시

```
interface
PropertyDescriptor {
    configurable?: boolean;
    enumerable?: boolean;
    value?: any;
    writable?: boolean;
    get?(): any;
    set?(v: any): void;
}
```

configurable: 기본값은 true. property의 property descriptor가 수정 가능한가
Ex) writable, enumerable, configurable이 수정 가능한가

enumerable: 기본값은 true. 열거 가능 여부. Ex) Object.keys로 열거 가능한가

value: 객체의 프로퍼티 값. Ex) method에서 property descriptor의 value는 method

writable: 기본값은 true. 프로퍼티의 수정 가능 여부

get: 객체에 접근할 때 실행될 함수

set: 객체를 수정할 때 실행될 함수

Property Descriptor

객체

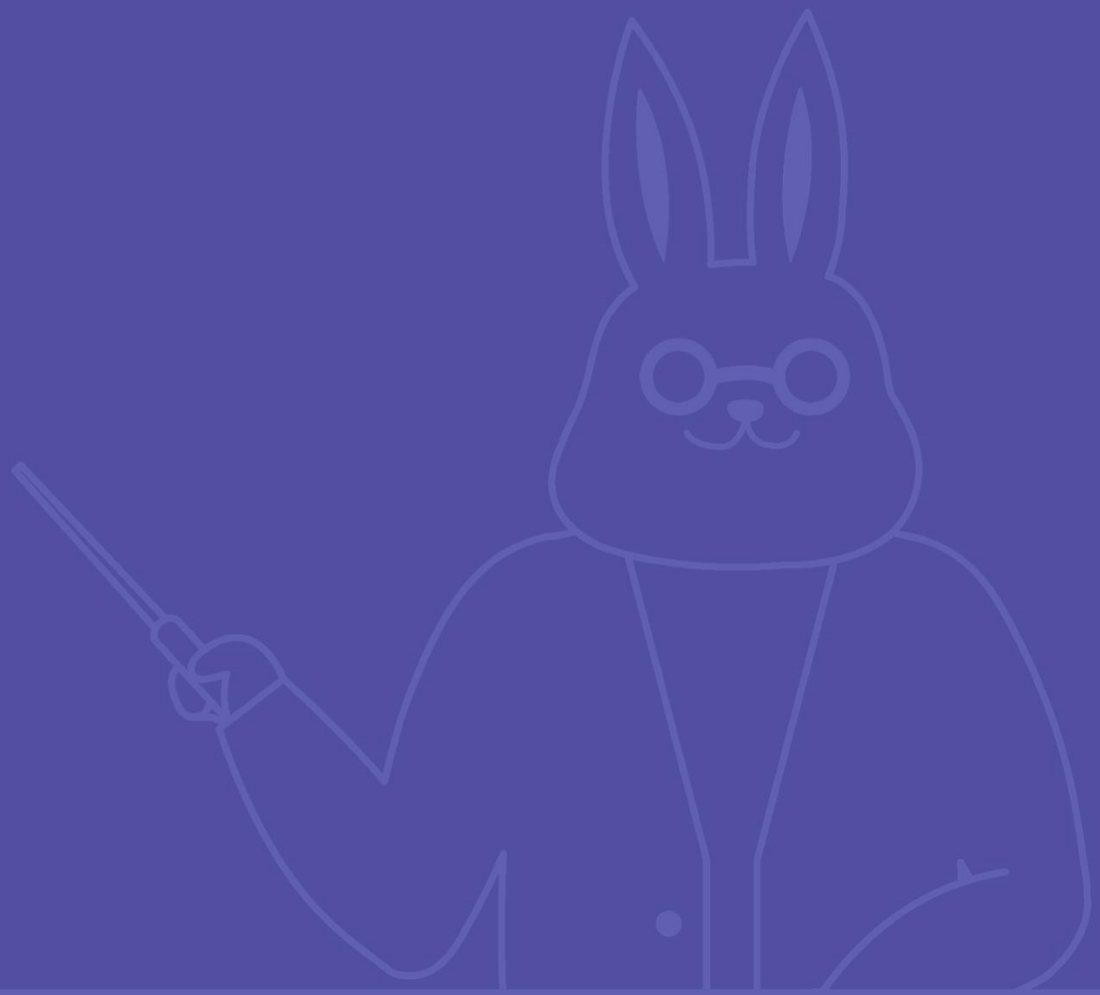
```
const human = {
  name: "elice",
  hello: function () {
    return "hello" + this.name;
  },
};
```

객체에 대한 Property Descriptor

```
const human = {
  name: "elice",
  hello: function () {
    return "hello" + this.name;
  },
};

const nameDescriptor =
Object.getOwnPropertyDescriptor(human, "name");
const helloResult = Object.getOwnPropertyDescriptor(human,
"hello")?.value.call(
  human
);
// {
//   value: 'elice',
//   writable: true,
//   enumerable: true,
//   configurable: true
// }
console.log(nameDescriptor);

// hello elice
console.log(helloResult);
```

03

Decorator Factory



Decorator를 사용하기 위한 tsconfig.json 설정

```
npm i reflect-metadata --save
```

```
yarn add reflect-metadata
```

tsconfig.json

```
{
  "compilerOptions": {
    "target": "ES5",
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true
  }
}
```

데코레이터는 아직 자바스크립트에 stage2에 있는 기능이므로 타입스크립트에서는 실험적 기능으로써 사용 가능.

따라서 데코레이터를 사용하려면 tsconfig.json에서 **experimentalDecorators 옵션을 true**로 설정해야 함.

데코레이터가 있는 선언에 대해 특정 타입의 메타 데이터를 내보내려면 **reflect-metadata 라이브러리 설치** 및 tsconfig.json에 **emitDecoratorMetadata 옵션을 true**로 설정.

Decorator Factory란?

고차 함수 = 함수를 반환하는 함수

팩토리 = 어떤 인스턴스를 생성하여 반환하는 함수, 메서드

데코레이터 팩토리 = **데코레이터 함수를 반환하는 함수**

Decorator Factory 용도

Enumerable 데코레이터

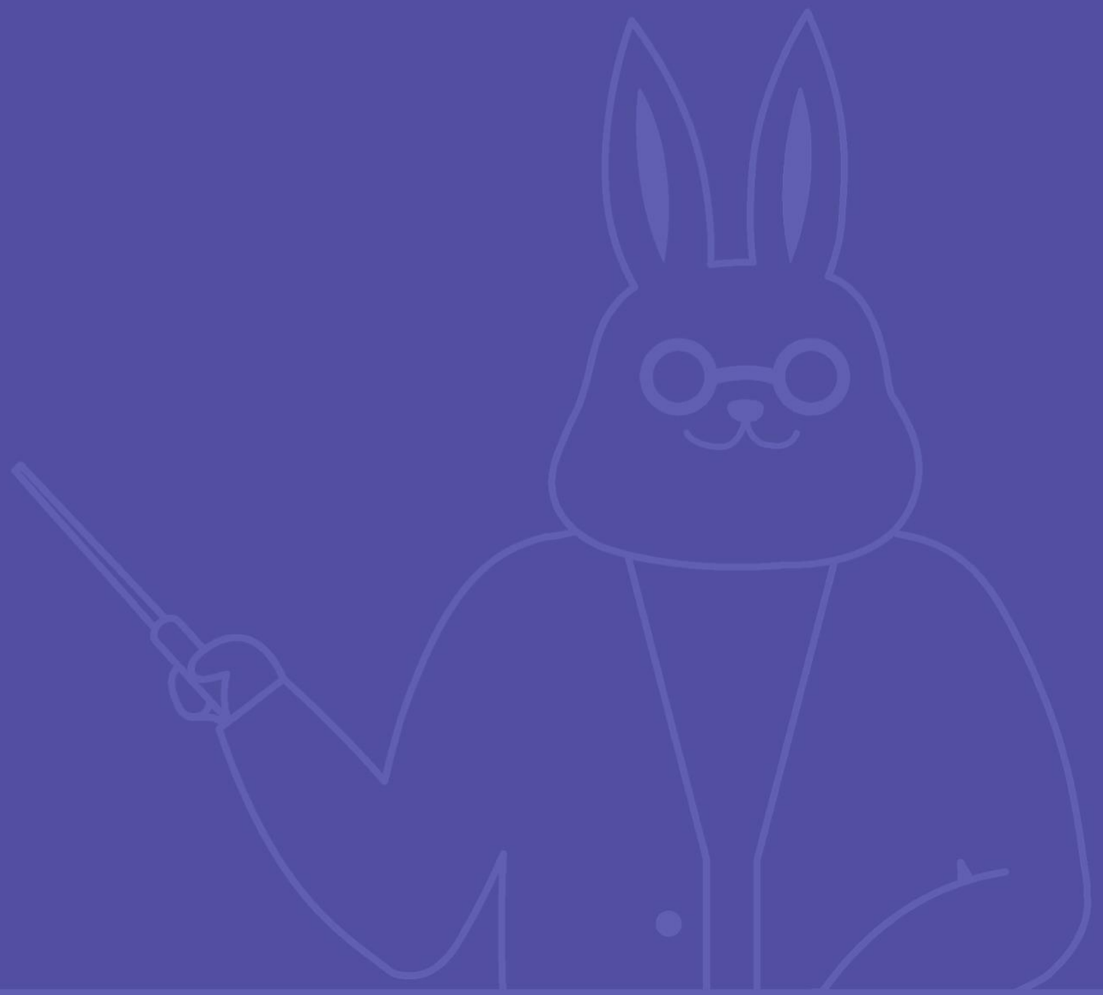
```
function enumerable(value: boolean) {
  return function (target: any,
propertyKey: string, descriptor:
PropertyDescriptor) {
    descriptor.enumerable = value;
  };
}

class Greeter {
  @enumerable(false)
  greet(name: string) {
    return "Hello, " + name;
  }
}
```

데코레이터 함수는 특정 인자만 받는다.

데코레이터에 추가적인 인자를 주고 싶다면 **고차함수에 매개변수를 추가하여 데코레이터 함수 내부에서 사용**할 수 있다.

04

Class Decorator



Class Decorator는?

클래스 데코레이터는

생성자 함수를 관찰, 수정, 오버라이딩(재정의)할 수 있다.

기존 클래스의 코드를 변경하지 않고 property를 추가하거나,

생성자 함수에서 기능을 추가하고 싶을 때 사용한다.

Class Decorator의 특징

Class Decorator 타입

```
// lib.es5.d.ts
declare type ClassDecorator
=  <TFunction extends
Function>(target: TFunction)
=> TFunction | void;
```

- 클래스 데코레이터는 클래스 선언 직전에 선언된다.
- 클래스 데코레이터는 런타임에 함수로서 호출된다.
- 클래스 데코레이터는 생성자 함수에 적용된다.
- 클래스 데코레이터의 target 매개변수는 생성자 함수이다.

Class Decorator의 용도

값을 반환하지 않는 경우

```
function sealed(constructor: Function)
{
  Object.seal(constructor);
  Object.seal(constructor.prototype);
}

@sealed
class Post {
  constructor(public title: string) {}

  publish(createdAt: Date) {
    console.log(`'${this.title}' is
published at
${createdAt.toISOString()}`);
  }
}
```

값을 반환하지 않는 경우 클래스의 기존 생성자 함수를 그대로 사용한다.

*Object.seal: 객체가 가진 기존 property를 수정하는 것 외에 다른 property를 추가하거나, 기존 property를 삭제하는 등의 변경은 가할 수 없게 만듦.

https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Global_Objects/Object/seal

Class Decorator의 용도

값을 반환하는 경우

```
function publishable<T extends { new
(...args: any[]): {} }>(constructor: T) {
  return class extends constructor {
    publishingURL = "http://www.elice.io";
  };
}

@publishable
class Post {
  constructor(public title: string) {}

  publish() {
    console.log(`'${this.title}' is
published on ${this.publishingURL}`);
  }
}
```

데코레이터가 값을 반환하면, 그 **반환값으로 생성자 함수를 교체**(재정의).

기존 생성자 함수에 기능을 확장하고 싶다면 **class extends constructor와 같이 생성자 함수를 확장하여 반환**

Class Decorator 사용 시 주의점

property를 추가해도 typescript는 알지 못 함

```
function publishable<T extends { new
(...args: any[]): {} }>(constructor: T) {
  return class extends constructor {
    publishingURL = "http://www.elice.io";
  };
}

@publishable
class Post {
  constructor(public title: string) {}

  publish() {
    // Property 'publishingURL' does not
exist on type 'Post'.
    console.log(`'${this.title}' is
published on ${this.publishingURL}`);
  }
}
```

데코레이터는 런타임에 호출되기 때문에

데코레이터로 class에 property를

추가해도 type system에서는 알지 못 함.

Decorator 사용 시 주의점

declare class에서 사용 시 에러

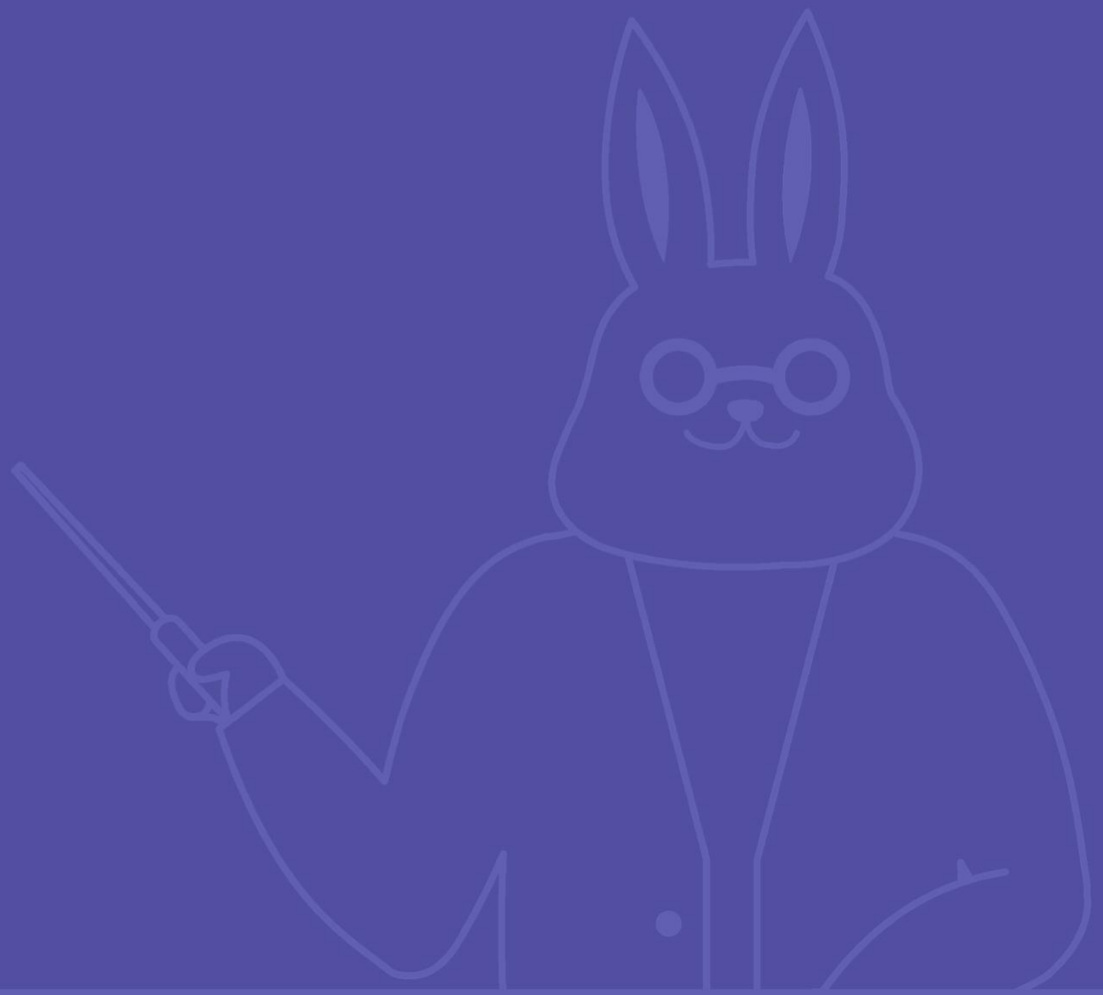
```
import { sealed } from "./decorators";

// Set the 'experimentalDecorators'
option in your
// 'tsconfig' or 'jsconfig' to remove
this warning
@sealed
declare class Post {
  title: string;
  desc: string;
}
```

d.ts 파일(타입 선언 파일), declare class에는 데코레이터 사용 못함.

tsconfig.json의 experimentalDecorators을 true로 설정해주어도 주석과 같은 에러 발생.

05

Method Decorator



Method Decorator는?

메서드 데코레이터는

메서드를 관찰, 수정, 오버라이딩(재정의)할 수 있다.

메서드의 Property descriptor를 수정하거나

기존 메서드 앞 뒤로 기능을 추가하고 싶을 때 사용할 수 있다.

(로깅, 타이머(performance), permission guard, etc.)

Method Decorator의 특징

Method Decorator 타입

```
// lib.es5.d.ts
interface TypedPropertyDescriptor<T>
{
  enumerable?: boolean;
  configurable?: boolean;
  writable?: boolean;
  value?: T;
  get?: () => T;
  set?: (value: T) => void;
}

declare type MethodDecorator = <T>(
  target: Object,
  propertyKey: string | symbol,
  descriptor:
TypedPropertyDescriptor<T>
) => TypedPropertyDescriptor<T> |
void;
```

- 메서드 데코레이터는 메서드 선언 직전에 선언된다.
- 메서드의 property descriptor에 적용된다.
- 메서드 데코레이터는 세 가지 인자를 받는다.

- target: 정적 멤버에 대한 클래스의 생성자 함수 또는 인스턴스 멤버에 대한 클래스의 프로토타입 (any, Object 타입)
- propertyKey: 메서드의 이름 (string, symbol 타입)
- descriptor: 메서드의 프로퍼티 설명자 (PropertyDescriptor)

Method Decorator의 용도

메서드 교체

```
function log() {
  return function ( target: any, property: string, descriptor:
PropertyDescriptor ) {
    const actualFunction = descriptor.value;

    const decoratorFunc = function (this: any, ...args: any[]) {
      console.log('${property} - start');
      const result = actualFunction.apply(this, args);
      console.log('${property} - end');
      return result;
    };

    descriptor.value = decoratorFunc;
    return descriptor;
  },
}

class Human {
  constructor(private firstName: string, private lastName: string) {}

  @log()
  introduce() {
    console.log('hello, my name is ${this.firstName} ${this.lastName}');
  }
}

const elice = new Human("elice", "song");
elice.introduce();
```

데코레이터가 값을 반환하면, 그 **반환값으로 propertyDescriptor이 교체된다.**

메서드를 교체하고 싶다면 descriptor.value에 새 함수를 할당

“introduce - start”
“hello, my name is elice song”
“introduce - end”

Method Decorator 사용 시 주의점

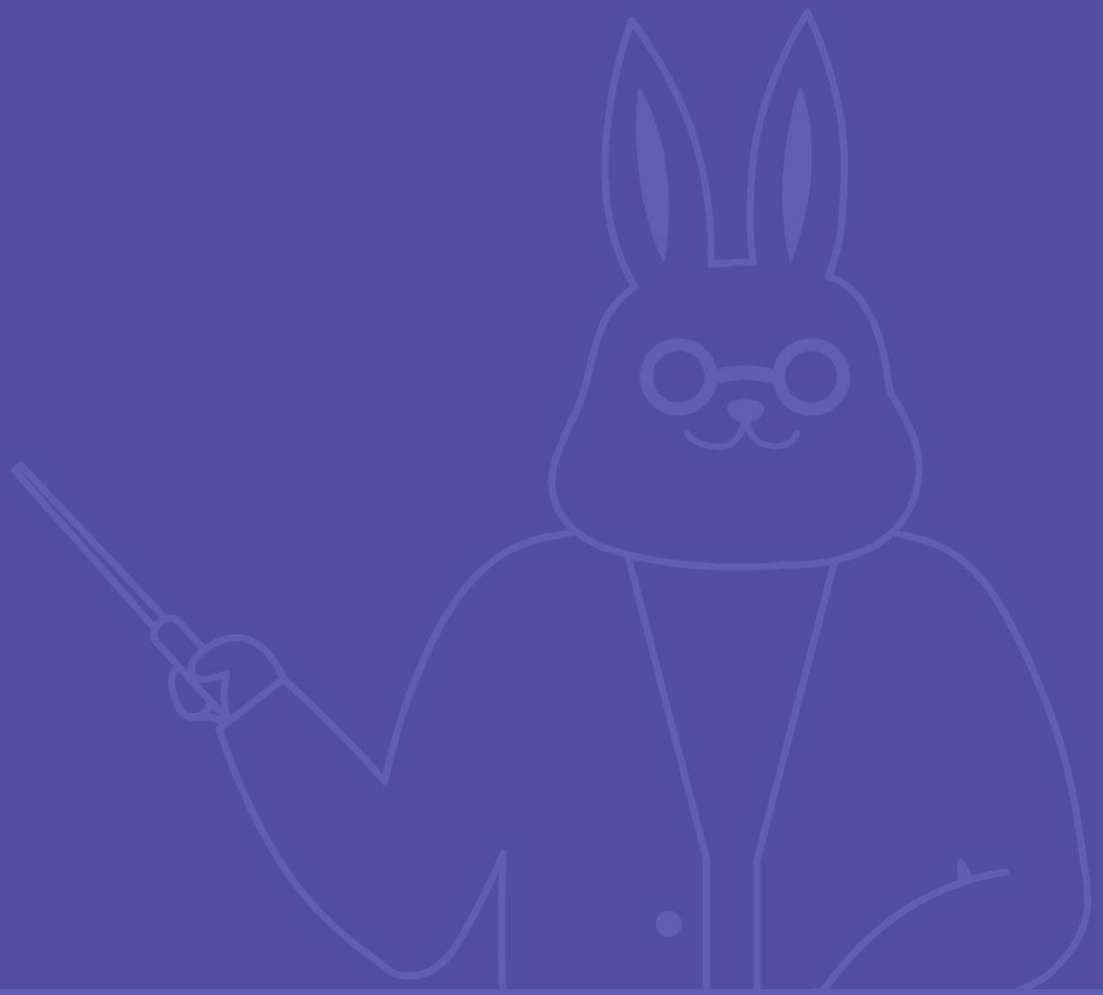
tsconfig.json에서 target을 es3로 설정

```
class Human {
  constructor(...) {}

  // Unable to resolve
signature of method decorator
  // when called as an
expression.
  @log()
  introduce() {
    ...
  }
}
```

- compilerOptions에서 target이 ‘ES5’ 보다 낮은 경우 property descriptor는 ‘undefined’이 된다.
- 또한, 반환값도 무시된다.
- 메서드 데코레이터 시그니처를 이해하지 못한다.

06

Accessor Decorator



Accessor Decorator는?

접근자(getter, setter) 데코레이터는

접근자를 관찰, 수정, 오버라이딩(재정의)할 수 있다.

접근자의 프로퍼티 설명자(Property Descriptor)에 적용된다.

전달받는 매개변수가 메서드 데코레이터와 똑같다.

Accessor Decorator의 용도

값을 반환하는 경우

```
function validate(regexp: string | RegExp) {
  return function ( target: any, property: string, descriptor:
PropertyDescriptor ) {
    const actualFunction = descriptor.set;

    const decoratorFunc = function (this: any, value: string) {
      const regexpTarget =
        typeof regexp === "string" ? new RegExp(regexp) : regexp;
      if (!regexpTarget.test(value)) {
        throw new Error("invalid value");
      }
      return actualFunction?.call(this, value);
    };

    descriptor.set = decoratorFunc;
  };
}

class Human {
  constructor() {}

  _phoneNumber = "";

  @validate(/010-[0-9]{3,4}-[0-9]{4}/)
  set phoneNumber(value: string) {
    this._phoneNumber = value;
  }
}
```

descriptor의 get 또는 set에 새로운 함수를 할당하여 getter, setter 접근자를 수정 또는 교체할 수 있다.

Accessor Decorator의 주의점

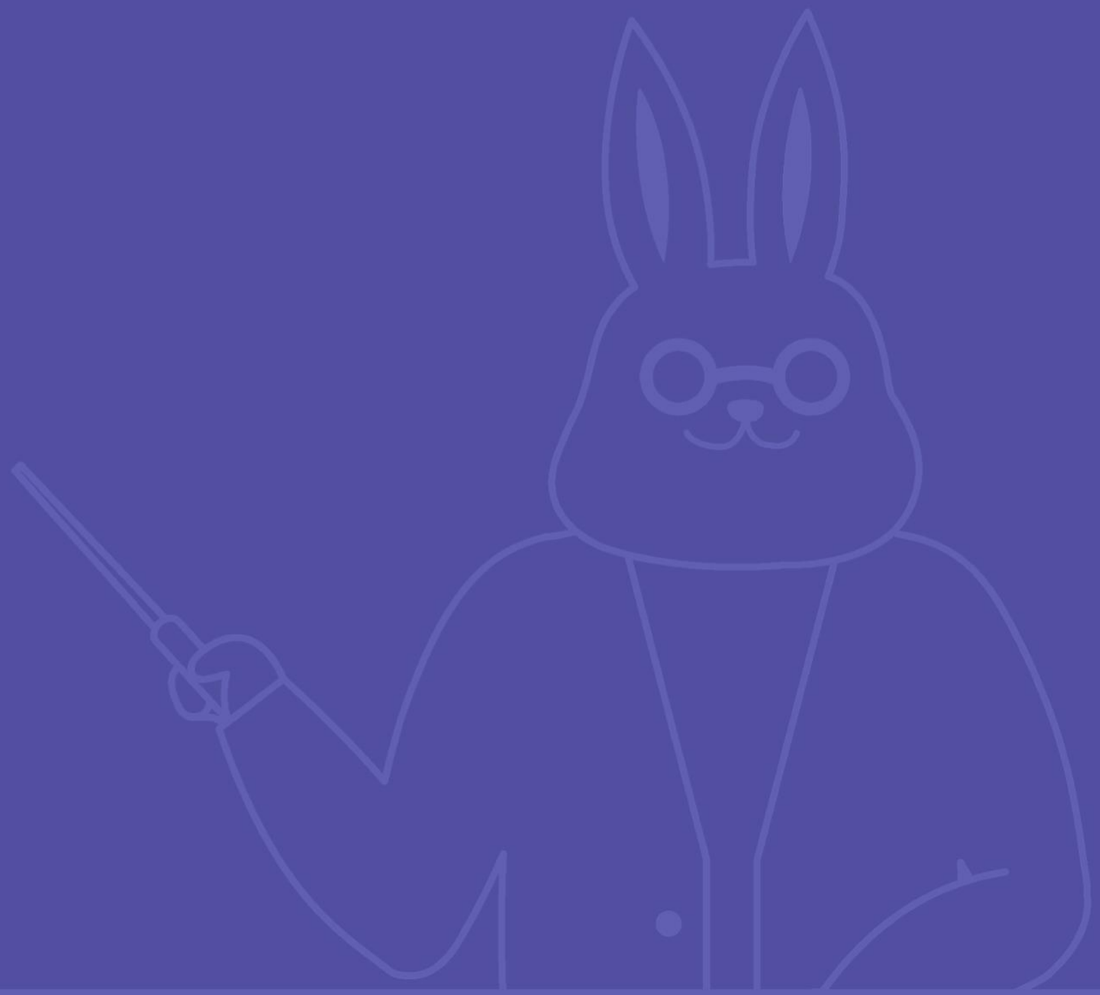
tsconfig.json에서 target을 es3로 설정

```
class Human {
  constructor() {}

  _phoneNumber = "";

  @validate(/010-[0-9]{3,4}-[0-9]{4}/)
  get phoneNumber() {
    return this._phoneNumber;
  }

  // Decorators cannot be applied to
multiple get/set accessors of the same name.
  @validate(/010-[0-9]{3,4}-[0-9]{4}/)
  set phoneNumber(value: string) {
    this._phoneNumber = value;
  }
}
```

- compilerOptions에서 target이 ‘ES5’보다 낮은 경우 property descriptor는 ‘undefined’이 된다.
- 또한, 반환값도 무시된다.
- 같은 프로퍼티에 대한 **접근자 get, set 중 하나에만 데코레이팅 가능**

07

Property Decorator



Property Decorator는?

프로퍼티 데코레이터는

프로퍼티를 관찰, 수정, 오버라이딩(재정의)할 수 있다.

프로퍼티의 Property descriptor를 수정하고 싶을 때 사용할 수 있다.

Property Decorator의 특징

Property Decorator 타입

```
declare type
PropertyDecorator = (
  target: Object,
  propertyKey: string | symbol
) => void |
PropertyDescriptor;
```

- 프로퍼티 데코레이터는 프로퍼티 선언 직전에 선언된다.
- **인스턴스가 아닌 클래스 프로토타입의 프로퍼티에 적용**된다.
- 프로퍼티 데코레이터는 target, propertyKey 두 가지 인자를 받는다.

*property descriptor를 매개변수로 하지 않는 이유:
프로퍼티의 경우, 인스턴스화되기 전까지 프로퍼티가 초기화되는 것을 관찰하거나 수정할 수 없음. 때문에 프로퍼티 데코레이터는 프로퍼티가 ‘클래스’에 선언되었음을 관찰하는 데만 사용할 수 있다.

Property Decorator의 용도

프로토타입 공유

```
function 영혼교환식(target: any, propertyKey:
string) {
  let name: string;

  const getter = function () {
    return name;
  };

  const setter = function (value: string) {
    name = value;
  };

  return {
    get: getter,
    set: setter,
    enumerable: true,
    configurable: true,
  } as any;
}
```

```
class Human {
  @영혼교환식
  public name: string;

  constructor(name: string) {
    this.name = name;
  }
}

const 햄스터 = new Human("햄스터");
const 그녀 = new Human("그녀");

그녀.name = "민석씨";
console.log(햄스터.name); // 민석씨
```

Property Decorator의 용도

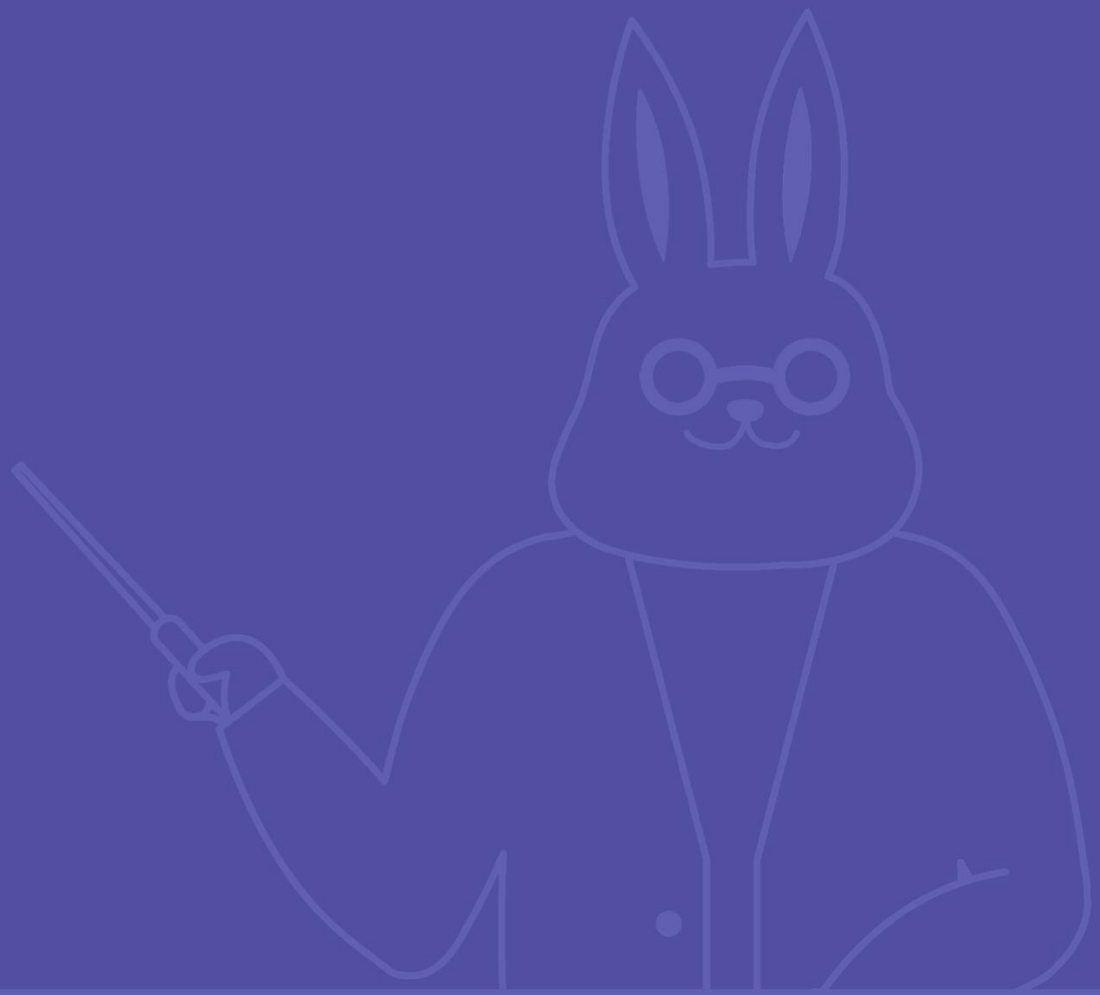
Symbol을 사용한 인스턴스 별 데코레이팅

```
function 영혼교환식(target: any,
propertyKey: string) {
  const uniqueKey = Symbol();

  const getter = function (this: any) {
    return this[uniqueKey];
  };

  const setter = function (this: any,
value: string) {
    this[uniqueKey] = value;
  };

  return {
    get: getter,
    set: setter,
    enumerable: true,
    configurable: true,
  } as any;
}
```

```
class Human {
  @영혼교환식
  public name: string;

  constructor(name: string) {
    this.name = name;
  }
}

const 햄스터 = new Human("햄스터");
const 그녀 = new Human("그녀");

그녀.name = "민석씨";
console.log(햄스터.name); // 햄스터
```

08

Parameter Decorator



Parameter Decorator는?

파라미터(매개변수) 데코레이터는

클래스의 생성자 함수 또는 메서드 선언 함수에 적용된다.

매개변수가 메서드에서 선언되었다는 것을 관찰하는 데에만 사용 가능

그 이상은 reflect-metadata 라이브러리 사용해야 함.

Parameter Decorator의 특징

Parameter Decorator 타입

```
declare type
ParameterDecorator = (
  target: Object,
  propertyKey: string |
symbol,
  parameterIndex: number
) => void;
```

- 파라미터 데코레이터는 매개변수 선언 직전에 선언된다.
- 파라미터가 선언된 함수에 적용된다.
- 파라미터 데코레이터는 세 가지 인자를 받는다.
- 파라미터 데코레이터의 반환값은 무시된다.

- target: 정적 멤버에 대한 클래스의 생성자 함수 또는 인스턴스 멤버에 대한 클래스의 프로토타입 (any, Object 타입)
- propertyKey: 멤버의 이름 (string, symbol 타입)
- **parameterIndex**: 함수의 매개변수 목록에 있는 매개변수의 인덱스. (서수 색인, ordinal index. 순서를 나타내는 인덱스를 말함)

Parameter Decorator의 용도

관찰

```
function watch(target: Object,
propertyKey: string, parameterIndex:
number) {
  console.log(
    `메서드 이름: ${propertyKey},
데코레이터 인덱스: ${parameterIndex}`
  );
}

class Human {
  constructor(public name: string) {}

  // "메서드 이름: hello, 데코레이터
인덱스: 0" 출력
  hello(@watch name: string) {}
}
```

reflect-metadata 없이는 메서드에서 선언되었다는 것만 관찰할 수 있음.

Parameter Decorator의 용도

수정, 교체

```
import "reflect-metadata";
const requiredMetadataKey = Symbol("required");

function required(target: Object, propertyKey: string | symbol,
parameterIndex: number) {
  let existingRequiredParameters: number[] =
Reflect.getOwnMetadata(requiredMetadataKey, target, propertyKey) || [];
  existingRequiredParameters.push(parameterIndex);
  Reflect.defineMetadata( requiredMetadataKey,
existingRequiredParameters, target, propertyKey);
}

function validate(target: any, propertyName: string, descriptor:
TypedPropertyDescriptor<Function>) {
  let method = descriptor.value!;

  descriptor.value = function () {
    let requiredParameters: number[] =
Reflect.getOwnMetadata(requiredMetadataKey, target, propertyName);
    if (requiredParameters) {
      for (let parameterIndex of requiredParameters) {
        if (parameterIndex >= arguments.length
          || arguments[parameterIndex] === undefined) {
          throw new Error("Missing required argument.");
        }
      }
    }
    return method.apply(this, arguments);
  };
}
```

수정, 교체

```
class BugReport {
  type = "report";
  title: string;

  constructor(t: string) {
    this.title = t;
  }

  @validate
  print(@required verbose: boolean) {
    if (verbose) {
      return this.title;
    } else {
      return `type: ${this.type}\ntitle:
${this.title}`;
    }
  }
}
```

Code: https://www.typescriptlang.org/docs/handbook/decorators.html

Parameter Decorator의 용도

수정, 교체

reflect-metadata를 통해 파라미터에 대한 메타데이터 수집, 저장.

메서드 데코레이터나 클래스 데코레이터와 함께 사용할 수 있음.

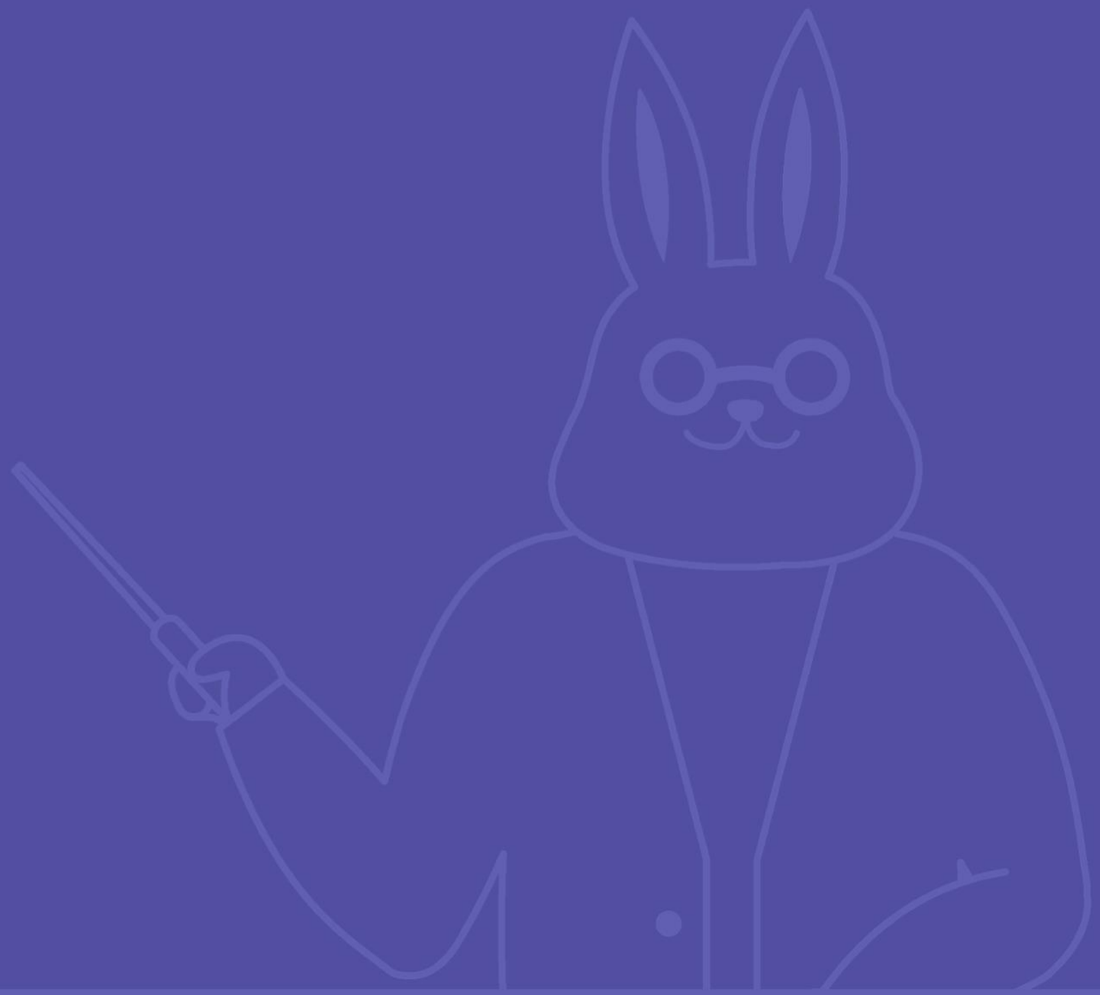
수정, 교체

```
class BugReport {
  type = "report";
  title: string;

  constructor(t: string) {
    this.title = t;
  }

  @validate
  print(@required verbose: boolean) {
    if (verbose) {
      return this.title;
    } else {
      return `type: ${this.type}\ntitle:
${this.title}`;
    }
  }
}
```

Code: https://www.typescriptlang.org/docs/handbook/decorators.html

09

Decorator의 동작



Decorator의 합성

Parameter Decorator 타입

```
class Human {
  constructor() {}

  @log()
  @timer()
  hello(name: string) {}
}

const elice = new Human();
elice.hello("elice");
```

여러 개의 데코레이터를 한 번에 쓰는 것

합성 함수 g(f(x))와 같은 매커니즘

log(timer())

timer 실행 후 log 실행

Decorator의 평가와 적용 순서

Parameter Decorator 타입

```
class Human {
  constructor() {}

  @log()
  @timer()
  hello(name: string) {}
}

const elice = new Human();
elice.hello("elice");
```

평가(선언): 위에서 아래로

적용(호출): 아래에서 위로

평가(선언): log -> timer

적용(호출): timer -> log

Decorator의 동작 원리: 컴파일된 _decorate 함수

조금 더 쉽게 나타낸 _decorator 함수

```
/**인자의 개수에 따라 descriptor 또는 target 반환*/
function getDescriptorOrTarget(target, desc, argumentsLength) {
  // ...
}

var __decorate = function (decorators, target, key, desc) {
  var argumentsLength = arguments.length;
  var descriptorOrTarget = getDescriptorOrTarget(target, desc,
argumentsLength);
  var decorator;
  // Reflect.decorate를 쓸 수 있으면 호출
  if (typeof Reflect === "object" && typeof Reflect.decorate ===
"function") {
    descriptorOrTarget = Reflect.decorate(decorators, target, key,
desc);
  } else {
    // 쓸 수 없다면 인자의 개수에 따라 매개변수를 다르게 주어 데코레이터 호출
    // 역방향으로 호출
    for (var i = decorators.length - 1; i >= 0; i--)
      if ((decorator = decorators[i]))
        descriptorOrTarget =
          (argumentsLength < 3
            ? decorator(descriptorOrTarget)
            : argumentsLength > 3
            ? decorator(target, key, descriptorOrTarget)
            : decorator(target, key)) || descriptorOrTarget;
  }
  // 후략...
};
```

- 먼저 Reflect.decorate를 지원하는 환경인지 체크.
- 지원하면 Reflect.decorate를 사용. 그렇지 않으면 타입스크립트에서 구현한 함수 사용.
- 데코레이터가 들어온 순서의 반대로 호출됨에 유의.
- 하나의 선언에 여러 개의 데코레이터가 적용될 때 합성 함수의 매커니즘처럼 작용하기 때문.

크레딧

/* elice */

코스 매니저
이재성

콘텐츠 제작자
송현지

강사
송현지

감수자
이재성

디자이너
김루미

연락처

TEL
070-4633-2015

WEB
https://elice.io

E-MAIL
contact@elice.io