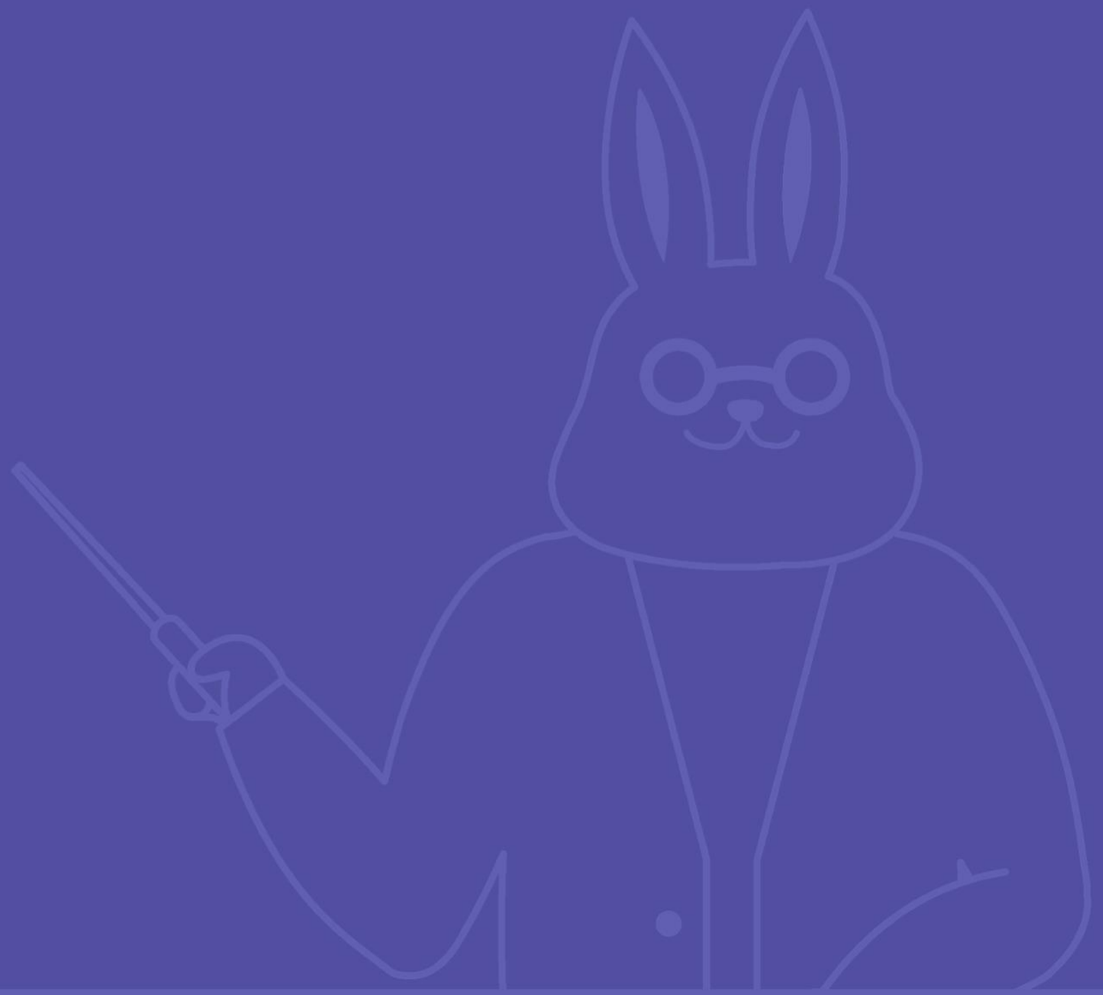

# 타입스크립트 I

## 02 Class



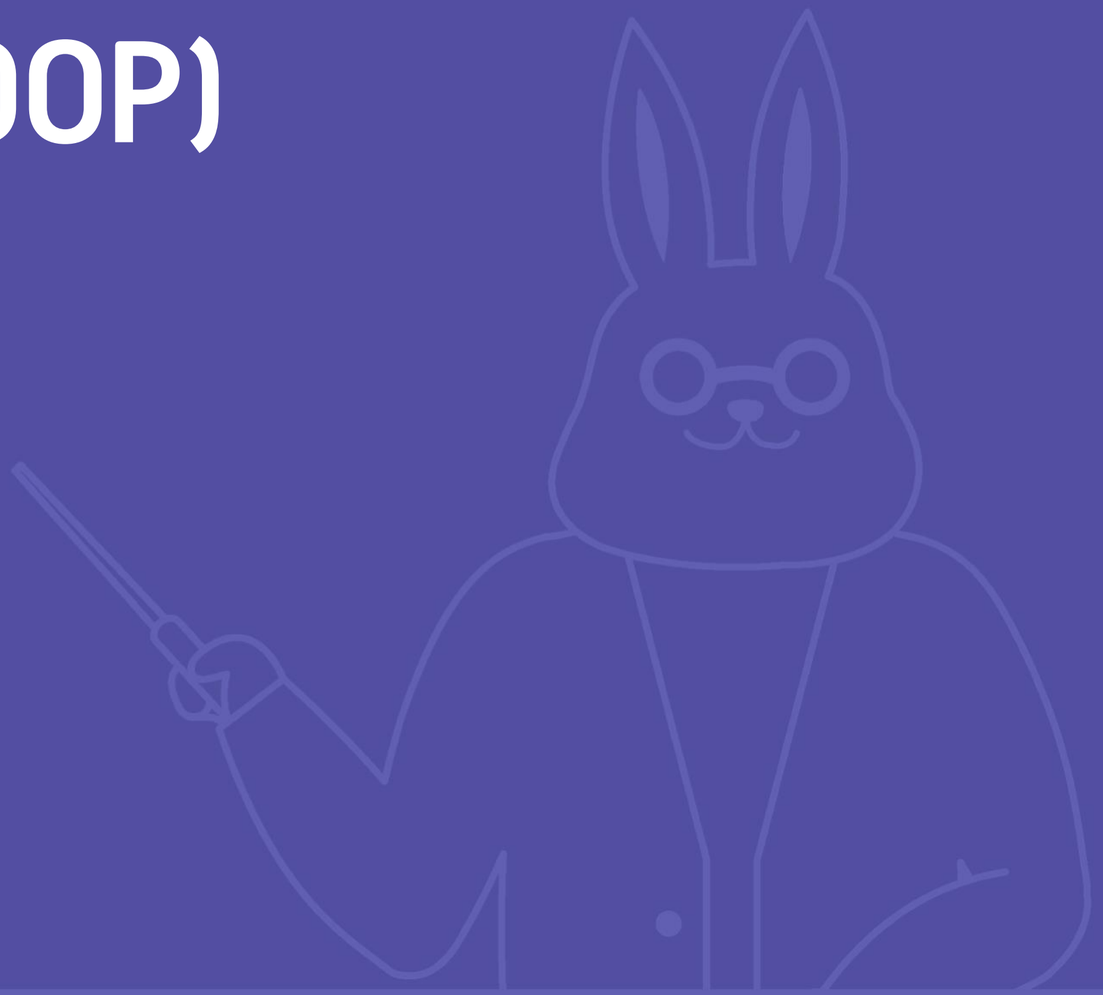


/* elice */

# 목차

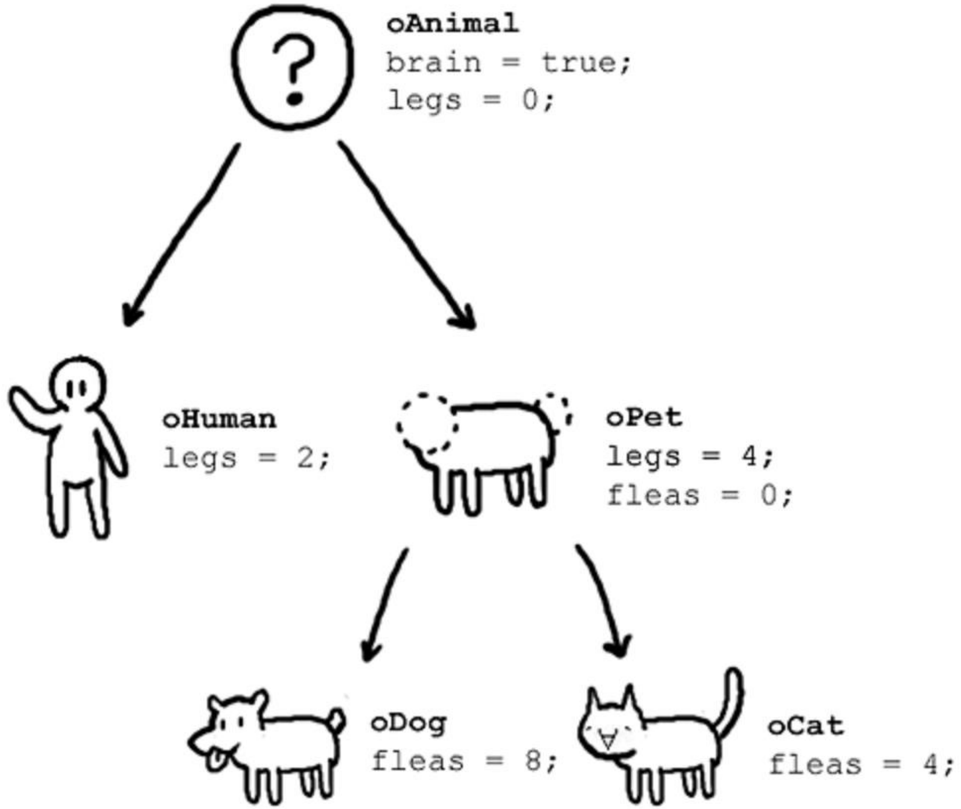
01

# 객체 지향 프로그래밍(OOP)



## 객체 지향 프로그래밍(OOP)이란?

- OOP는 컴퓨터 프로그램을 **객체(Object)**의 모임으로 파악하려는 프로그래밍 패러다임이다.
- **객체(Object)**들은 서로 메시지를 주고 받을 수 있으며 데이터를 처리할 수 있다.



## 객체 지향 프로그래밍(OOP)의 장점

- 프로그램을 유연하고 변경이 용이하게 만든다.
- 프로그램의 개발과 보수를 간편하게 만든다.
- 직관적인 코드 분석을 가능하게 한다.

- 객체 지향 프로그래밍의 중요한 특성

  *강한 응집력(Strong Cohesion)*과 *약한 결합력(Weak Coupling)*을 지향한다.

## Class 용어 설명

- 클래스의 요소
  멤버(member)
  필드(field)
  생성자(constructor)
  메소드(method)

- 인스턴스(instance) : new 연산자에 의해서 생성된 객체

## Class 생성하기

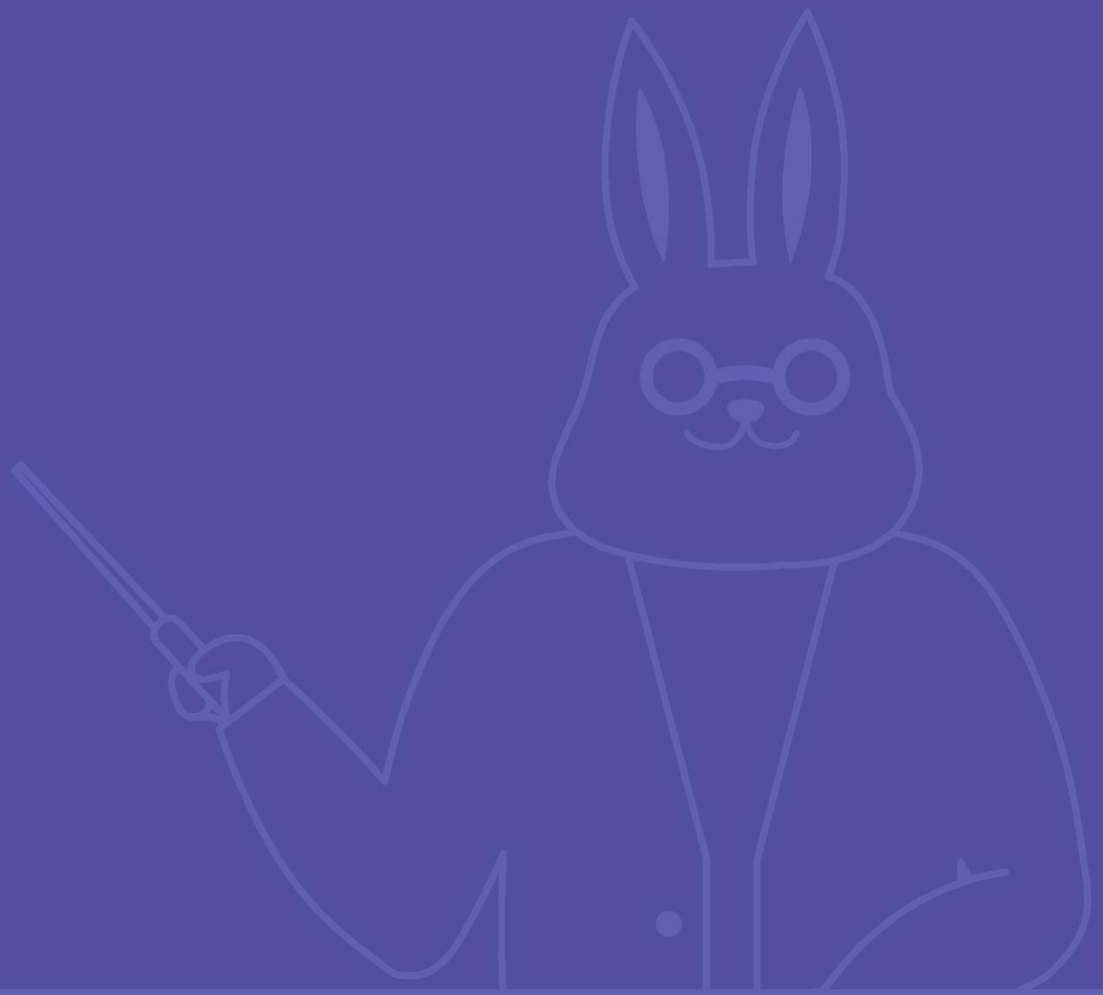
- **new**를 사용하여 Person 클래스의 인스턴스를 생성한다.
- Person class의 멤버는 name, constructor, say()가 있다.
- 클래스 안에서 “this.”를 앞에 붙이면 클래스의 멤버를 의미한다.

코드

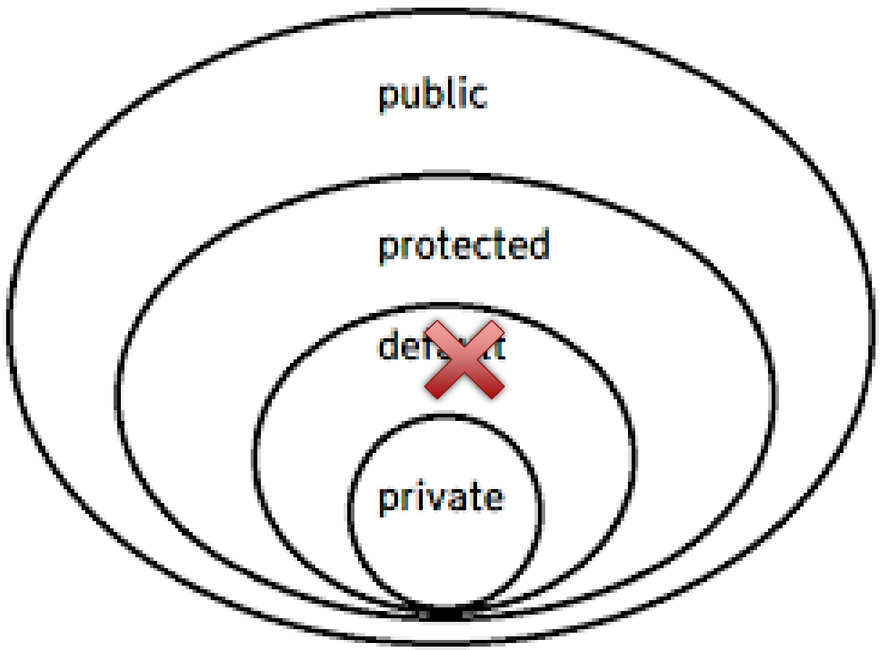
```
class Person {
  name: string;
  constructor(name: string) {
   this.name = name;
  }
  say() {
   return "Hello, My name is " + this.name;
  }
}

let person = new Person("june");
```

02

# 접근 제어자 / 상속



## 접근 제어자




- 속성 또는 메소드로의 접근을 제한하기 위해 사용한다.
- TypeScript에는 3종류의 접근 제어자가 존재한다.
- public > protected > private
- Java와 다르게 package 개념이 없어 default 접근 제어자는 존재하지 않는다.

## 접근 제어자(public)

- 프로그램 내에서 선언된 멤버들이 자유롭게 접근할 수 있다.
- TypeScript에서 멤버는 기본적으로 public으로 선언된다.
- 명시적으로 멤버를 public으로 표시할 수도 있다.

코드

```
class Animal {
  public name: string
  constructor(theName: string) {
    this.name = theName;
  }
}
new Animal("Cat").name ;
```

## 접근 제어자(private)

- 멤버가 포함된 클래스 외부에서의 접근을 막는다.

코드

```
class Animal {
  private name: string
  constructor(theName: string) {
    this.name = theName;
  }
}
new Animal("Cat").name // Error: Property 'name' is private and only accessible within class 'Animal'
```

## 상속

- OOP는 상속을 이용하여 존재하는 클래스를 확장해 새로운 클래스를 생성할 수 있다.
- extends 키워드로 Animal이라는 기초 클래스에서 Dog 클래스가 파생되었다.
- 파생된 클래스는 하위클래스(subclass), 기초 클래스는 상위클래스(superclass)라고 부른다.
- Dog, Cat은 Animal의 기능(move 메소드)을 확장하기 때문에, move()와 makeDound()를 가진 인스턴스를 생성한다.

예시

```
class Animal {
  move(distanceInMeters: number) {
    console.log(`Animal moved ${distanceInMeters}m.`);
  }
}

class Dog extends Animal {
  makeSound() {
    console.log("멍멍!");
  }
}

class Cat extends Animal {
  makeSound() {
    console.log("야옹!");
  }
}

const dog = new Dog();
dog.move(10);
dog.makeSound();

const cat = new Cog();
cat.move(5);
cat.makeSound();
```

## 접근 제어자(protected)

- 멤버가 포함된 클래스와 그 하위 클래스를 제외한 외부에서의 접근을 막는다.
- Person에서 파생된 Employee의 인스턴스 메소드에서는 name을 사용할 수 있다.

## 접근 제어자(protected)

코드

```
class Person {
  protected name: string
  constructor(name: string) {
    this.name = name;
  }
}

class Employee extends Person {
  private department: string

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

  public getElevatorPitch() {
    return 'Hello, my name is ${this.name} and I work in ${this.department}.';
  }
}

let howard = new Employee("Howard", "Sales");
console.log(howard.getElevatorPitch());
console.log(howard.name); // Error: Property 'name' is protected and only accessible within class 'Person' and its subclasses.
```

출처 - https://github.com/microsoft/TypeScript

03

# Getters & Setters / readonly / static

## Getters & Setters / readonly / static

- getters & setters

  비공개로 설정하려는 속성은 private로 설정하고, 속성값을 읽고 수정하는 getter/setter 함수를 사용한다.

- readonly

  읽기만 가능한 속성을 선언하기 위해 사용한다.

- static

  전역 멤버를 선언할 때 사용한다.

  전역멤버 : 객체마다 할당되지 않고 클래스의 모든 객체가 공유하는 멤버

## Getters & Setters

- class의 속성에 직접 접근하는 것을 막고, getter, setter 함수를 사용해 값을 받아오거나 수정한다.
  속성에 직접 접근해 수정하면 데이터 무결성이 깨질 수 있다. (캡슐화 권장)
- 각 객체의 멤버에 접근하는 방법을 세밀하게 제어할 수 있다.

코드

```
class Person {
  private _name: string

  get name() {
    return this._name;
  }
  set name(name: string) {
    if (name.length > 10) {
      throw new Error("name too long")
    }
    this._name = name;
  }
}
```

```
let person = new Person();

console.log(person.name); // undefined
person.name = "june";
console.log(person.name); // june
person.name = "junejunejunejunejune"; // throw Error
```

## readonly

- 속성을 읽기 전용으로 설정해 변경할 수 없게 만든다.
- 선언될 때나 생성자에서 값을 설정하면 이후 수정할 수 없다.

코드

```
class Person
  readonly age: number = 20 // 선언 초기화
  constructor(age: number) {
    this.age = age;
  }
}

let person = new Person(10); // 생성자 초기화
person.age = 30; // Error: Cannot assign to 'age' because it is a read-only property.
```

## static

- 각 인스턴스가 아닌 클래스 자체에서 보이는 전역 멤버를 생성한다.
- 범용적으로 사용되는 값에 설정한다.
- “클래스명.”을 앞에 붙여 static 멤버에 접근할 수 있다.
- ES6에서는 메소드 전용 속성에는 선언이 안되었으나, TypeScript에서는 사용할 수 있다.

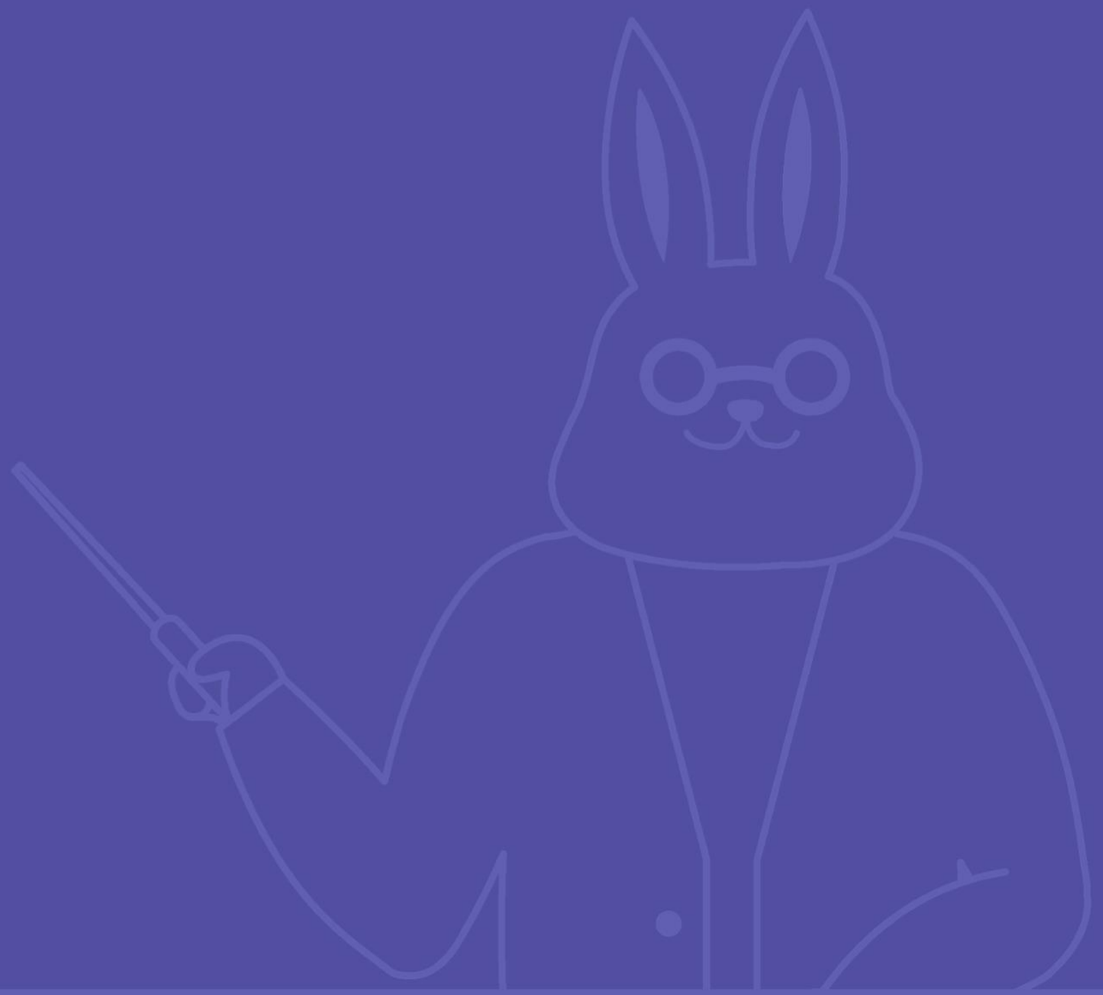
코드

```
class Grid {
  static origin = { x: 0, y: 0 }
  calculateDistanceFromOrigin(point: { x: number; y: number }) {
    let xDist = point.x - Grid.origin.x;
    let yDist = point.y - Grid.origin.y;
    return Math.sqrt(xDist * xDist + yDist * yDist) / this.scale;
  }
  constructor(public scale: number) {}
}

let grid1 = new Grid(1.0); // 1x scale
let grid2 = new Grid(5.0); // 5x scale

console.log(grid1.calculateDistanceFromOrigin({ x: 10, y: 10 }));
console.log(grid2.calculateDistanceFromOrigin({ x: 10, y: 10 }));
```

출처 - https://github.com/microsoft/TypeScript

04

# 추상 클래스



## 추상 클래스

- 추상 클래스는 다른 클래스들이 파생될 수 있는 기초 클래스이다.
- 직접 인스턴스화 할 수 없다.
- **abstract** 키워드는 추상 클래스나 추상 메소드를 정의하는 데 사용된다.
- 추상 메소드는 클래스에는 구현되어 있지 않고, 파생된 클래스에서 구현해야 한다.

## 추상 클래스

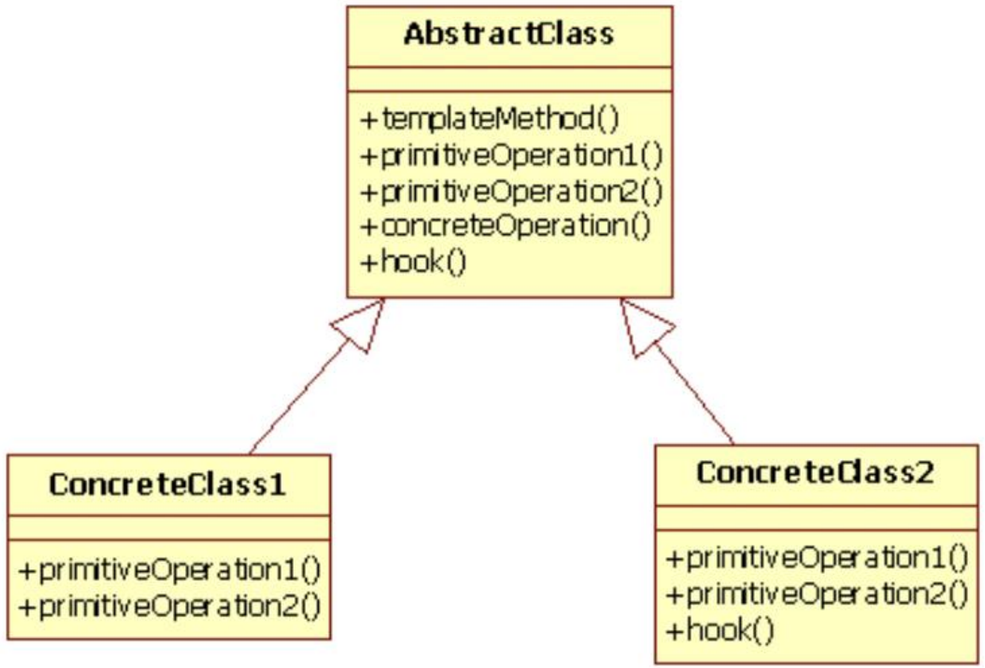
코드

```
abstract class Animal {
  protected name: string

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

  abstract makeSound(): void

  move(): void {
    console.log("move !!");
  }
}
```

```
class Dog extends Animal {
  constructor(name: string) {
    super(name); // 파생된 클래스의 생성자는 반드시 super()를 호출
  }
  makeSound(): void { // 반드시 파생된 클래스에서 구현 필수
    console.log(this.name + " 멍멍!!");
  }
}

const animal = new Animal("animal");
// Error: Cannot create an instance of an abstract class
const dog = new Dog("진돗개");

dog.makeSound(); // 진돗개 멍멍!!
```

## 추상 클래스를 활용한 디자인 패턴 (Template Method Pattern)

- 프로그램의 일부분을 서브 클래스로 캡슐화해 전체 구조를 바꾸지 않고 특정 단계의 기능을 바꾸는 것을 디자인 패턴이라고 한다.
- 전체적인 알고리즘은 상위 클래스에서 구현하고 다른 부분은 하위 클래스에서 구현한다.
- 전체 구조는 유사하지만 부분적으로 다른 구문으로 구성된 메소드의 코드 중복을 최소화 할 수 있다.



## 추상 클래스를 활용한 디자인 패턴 (Template Method Pattern)

코드

```
abstract class Parent {
  //템플릿 메소드: 자식에서 공통적으로 사용하는 부분(someMethod)
  public do() {
    console.log("Parent에서 실행 - 상");

    this.hook(); // 훅 메소드: Child에서 구현해야 할 부분

    console.log("Parent에서 실행 - 하");
  }
  abstract hook(): void
}

class Child extends Parent {
  hook(): void {
    console.log("Child");
  }
}
```

```
const child = new Child();
child.do();

// 실행 결과

// Parent에서 실행 - 상
// Child
// Parent에서 실행 - 하
```

# 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