TypeScript Class

TypeScript

TypeScript Class

1. JavaScript Class

JavaScript class는 ECMAScript 6을 통해 소개되었으며, 기존 prototype 기반의 상속 보다 명료하게 사용할 수 있다. Class 문법은 새로운 객체지향 상속 모델을 제공하는 것은 아니다. JavaScript class는 객체를 생성하고 상속을 다루는데 있어 훨씬 더 단순하고 명확한 문법을 제공한다.

Class 선언

class를 정의하는 한 가지 방법은 **class 선언**을 이용하는 것이다. class를 선언하기 위해서는 클래스의 이름과 함께 class 키워드를 사용한다.

```
class Polygon {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
}
```

TypeScript

TypeScript Class

2. TypeScript Class

TypeScript class는 JavaScript의 Class와 크게 다르지 않다. 다른 점은 Class 내부에서 사용할 멤버 변수를 정의해야 하며 필요에 따라 생성자를 이용하여 초기화를 진행한다. 멤버 변수는 필요에 따라 public 이외의 접근제한자를 할당할 수 있다.

Class 선언

class를 정의하는 방법은 JavaScript와 마찬가지로 **class 선언**을 이용하는 것이다. class를 선언하기 위해서는 클래스의 이름과 함께 class 키워드를 사용한다.

```
class Polygon {
    public width: number;
    public height: number;
    constructor(height: number, width: number) {
        this.height = height;
        this.width = width;
    }
}
```

TypeScript

TypeScript Class

3. Class 상속

TypeScript에서는 일반적인 객체 지향 패턴을 사용할 수 있다. 그 중에 클래스 기반 프로그래밍에서 가장 기본적인 패턴 중 하나인 상속을 사용하여 기존 클래스를 확장하여 새로운 클래스를 생성할 수 있다는 것이다.

생성자 함수를 포함하는 파생 클래스는 기본 클래스에서 생성자 함수를 실행할 때 무조건 super() 를 호출해야 한다. 또한 기본 클래스의 메서드를 하위 클래스에 특화된 메서드로 재정의 할 수 있다(overloading)

```
class Polygon {
    public height: number;
    public width: number
    constructor(height: number, width: number) {
        this.height = height;
        this.width = width;
    }
    public getArea(): void {
        console.log(this.height * this.width);
    }
}
```

```
class Triangel extends Polygon {
    constructor(height: number, width: number){
        super(height, width);
    }
    public getArea(): void {
        console.log(this.height * this.width / 2);
    }
}
```

TypeScript

TypeScript Modifier & Parameter property

4. 접근제한자

TypeScript에서는 접근제한자로 public, private과 protected를 사용할 수 있다.

public: 기본 접근제한자는 public으로 public이 붙은 멤버 변수, 함수는 어디서나 참조가 가능하다.

private: 멤버 변수나 함수가 private로 선언되면 클래스 외부에서는 참조할 수 없다. TypeScript는 구조형 시스템이다. 만약 두 가지 타입을 비교할 때, 그들이 어디서 왔는지에 관계없이, 모든 구성원의 타입이 호환 가능하다면 타입 자체가 호환 가능하다고 한다.

protected: protected 키워드는 private과 유사하게 동작한다. 단 protected로 선언된 멤버는 파생 클래스의 인스턴스에서 액세스 할 수 있습니다. 또한 생성자는 protected로 표시 될 수도 있다. 즉, 클래스 외부에서 클래스를 인스턴스화 할 수는 없지만 확장할 수는 있다.

readonly: readonly 키워드를 이용하여 읽기 전용 멤버 변수를 만들 수 있다. readonly 속성은 멤버 변수 또는 생성자에서 초기화해야 한다.

TypeScript

TypeScript Accessor

5. Getter / Setter (Accessor)

TypeScript는 객체의 멤버에 대한 액세스를 지원하는 Getter/Setter 메서드가 있다. 이 메서드를 이용하여 객체의 멤버 프로퍼티가 액세스되는 방식을 세밀하게 제어할 수 있다.

Getter/Setter는 ECMAScript 5 이상을 사용하도록 컴파일러를 설정해야 한다. ECMAScript 3에 대한 하위 레벨링은 지원되지 않는다.

```
private _kor: number

public set kor(kor: number){
	this._kor = kor;
}

public get kor(): number {
	return this._kor;
}
```

TypeScript

TypeScript Abstract Class

5. Abstract Class

추상 클래스는 다른 클래스를 파생시킬 수 있는 기본 클래스이다. 하지만 직접 인스턴스화 할 수 없다. 인터페이스와 달리 추상 클래스는 멤버에 대한 Implementation 세부 정보를 포함할 수 있다.
abstract 키워드는 추상 클래스 내 추상 메서드 뿐만 아니라 추상 클래스를 정의하는데 사용된다.

abstract 로 표시된 추상 클래스 내의 메소드에는 Implementation이 포함되어 있지 않으므로 파생 클래스에서 구현해야 한다. 추상 메소드는 인터페이스 메소드와 유사한 구문을 사용한다. 둘 다 메소드 본문을 포함하지 않고 method signature만 정의한다. 그러나 추상 메소드는 abstract 키워드를 포함해야 하며 선택적으로 Getter/Setter를 포함할 수 있다.

```
abstract class Jumsu {
        public readonly kor: number;
        public constructor(private name: string, kor: number, eng: number){ .. }
        public abstract getTotal(): number;
        public abstract getAvg(su: number): number;
}
```

TypeScript

TypeScript Interface

6. Interface

C# 및 Java와 같은 언어로 인터페이스를 사용하는 가장 일반적인 방법 중 하나는 클래스가 특정 계약을 준수하도록 명시적으로 적용하는 것이다. TypeScript에서도 가능하다. 인터페이스는 클래스의 public만 기술할 수 있다.
인터페이스는 여러 인터페이스를 확장하고 모든 인터페이스를 조합하여 만들 수 있다.

```
interface IJumsu {
        getTotal(): number;
        getAvg(num: number): number;
        display(): string;
}
abstract class AJumsu implements IJumsu{
        constructor(name: string, kor: number, eng: number){ … }
        public abstract getTotal(): number;
        public abstract getAvg(num: number): number;
}
class JumsuThree implements IJumsu { … }
```

TypeScript

TypeScript Interface

6. Interface의 확장

인터페이스는 여러 인터페이스를 확장하고 모든 인터페이스를 조합하여 만들 수 있다.

```
interface IJumsu {
    getTotal(): number;
    getAvg(num: number): number;
}
interface IDisplay {
    display(): string;
}

class JumsuThree implements Ijumsu, IDisplay { ... }
```