12.4 TypeScript - Class

클래스

ES6에서 새롭게 도입된 클래스는 기존 프로토타입 기반 객체지향 언어보다 클래스 기반 언어에 익숙한 개발자가 보다 빠르게 학습할 수 있는 단순명료한 새로운 문법을 제시하고 있다. 하지만 클래스가 새로운 객체지향 모델을 제공하는 것은 아니다. 사실 클래스도 함수이고 기존 프로토타입 기반 패턴의 Syntactic sugar일 뿐이다. Typescript가 지원하는 클래스는 ECMAScript 6의 클래스와 상당히 유사하지만 몇 가지 Typescript만의 고유한 확장 기능이 있다.

#1. 클래스 정의 (Class Definition)

ES6 클래스는 클래스 몸체에 메소드만을 포함할 수 있다. 클래스 몸체에 클래스 프로퍼티를 선언할 수 없고 반드시 생성자 내부에서 클래스 프로퍼티를 선언하고 초기화한다.

JAVASCRIPT

```
// person.js
class Person {
  constructor(name) {
    // 클래스 프로퍼티의 선언과 초기화
    this.name = name;
  }

  walk() {
    console.log(`${this.name} is walking.`);
  }
}
```

위 예제는 ES6에서 문제없이 실행되는 코드이지만 위 파일의 확장자를 ts로 바꾸어 Typescript 파일로 변경한 후, 컴파일하면 아래와 같이 컴파일 에러가 발생한다.

CODE

```
person.ts(4,10): error TS2339: Property 'name' does not exist on type 'Perso
n'.
person.ts(8,25): error TS2339: Property 'name' does not exist on type 'Perso
n'.
```

Typescript **클래스는 클래스 몸체에 클래스 프로퍼티를 사전 선언하여야 한다**.

TYPESCRIPT

```
// person.ts
class Person {
  // 클래스 프로퍼티를 사전 선언하여야 한다
  name: string;

  constructor(name: string) {
    // 클래스 프로퍼티수에 값을 할당
    this.name = name;
  }

  walk() {
    console.log(`${this.name} is walking.`);
```

```
  }
}

const person = new Person('Lee');
person.walk(); // Lee is walking
```

#2. 접근 제한자

Typescript 클래스는 클래스 기반 객체 지향 언어가 지원하는 접근 제한자(Access modifier) public, private, protected 를 지원하며 의미 또한 기본적으로 동일하다.

단, 접근 제한자를 명시하지 않았을 때, 다른 클래스 기반 언어의 경우, 암묵적으로 protected로 지정되어 패키지 레벨로 공개되지만 Typescript의 경우, 접근 제한자를 생략한 클래스 프로퍼티와 메소드는 암묵적으로 public이 선언된다. 따라서 public으로 지정하고자 하는 멤버 변수와 메소드는 접근 제한자를 생략한다.

접근 제한자를 선언한 프로퍼티와 메소드에 대한 접근 가능성은 아래와 같다.

접근 가능성	public	protected	private
클래스 내부	○	○	○
자식 클래스 내부	○	○	×
클래스 인스턴스	○	×	×

아래의 예제를 통해 접근 제한자가 선언된 프로퍼티로의 접근 가능성에 대해 살펴보자.

TYPESCRIPT

```
class Foo {
  public x: string;
  protected y: string;
  private z: string;

  constructor(x: string, y: string, z: string) {
    // public, protected, private 접근 제한자 모두 클래스 내부에서 참조 가능하다.
    this.x = x;
    this.y = y;
```

```
    this.z = z;
  }
}

const foo = new Foo('x', 'y', 'z');

// public 접근 제한자는 클래스 인스턴스를 통해 클래스 외부에서 참조 가능하다.
console.log(foo.x);

// protected 접근 제한자는 클래스 인스턴스를 통해 클래스 외부에서 참조할 수 없다.
console.log(foo.y);
// error TS2445: Property 'y' is protected and only accessible within class
 'Foo' and its subclasses.

// private 접근 제한자는 클래스 인스턴스를 통해 클래스 외부에서 참조할 수 없다.
console.log(foo.z);
// error TS2341: Property 'z' is private and only accessible within class 'F
oo'.

class Bar extends Foo {
  constructor(x: string, y: string, z: string) {
    super(x, y, z);

    // public 접근 제한자는 자식 클래스 내부에서 참조 가능하다.
    console.log(this.x);

    // protected 접근 제한자는 자식 클래스 내부에서 참조 가능하다.
    console.log(this.y);

    // private 접근 제한자는 자식 클래스 내부에서 참조할 수 없다.
    console.log(this.z);
    // error TS2341: Property 'z' is private and only accessible within clas
s 'Foo'.
  }
}
```

3. 생성자 파라미터에 접근 제한자 선언

접근 제한자는 생성자 파라미터에도 선언할 수 있다. 이때 **접근 제한자가 사용된 생성자 파라미터는 암묵적으로 클래스 프로퍼티로 선언되고 생성자 내부에서 별도의 초기화가 없어도 암묵적으로 초기화가 수행된다.**

이때 private 접근 제한자가 사용되면 클래스 내부에서만 참조 가능하고 public 접근 제한자가 사용되면 클래스 외부에서도 참조가 가능하다.

TYPESCRIPT

```
class Foo {
  /*
  접근 제한자가 선언된 생성자 파라미터 x는 클래스 프로퍼티로 선언되고 지동으로 초기화된다.
  public이 선언되었으므로 x는 클래스 외부에서도 참조가 가능하다.
  */
  constructor(public x: string) { }
}

const foo = new Foo('Hello');
console.log(foo);   // Foo { x: 'Hello' }
console.log(foo.x); // Hello

class Bar {
  /*
  접근 제한자가 선언된 생성자 파라미터 x는 멤버 변수로 선언되고 자동으로 초기화된다.
  private이 선언되었으므로 x는 클래스 내부에서만 참조 가능하다.
  */
  constructor(private x: string) { }
}

const bar = new Bar('Hello');

console.log(bar); // Bar { x: 'Hello' }

// private이 선언된 bar.x는 클래스 내부에서만 참조 가능하다
console.log(bar.x); // Property 'x' is private and only accessible within cl
ass 'Bar'.
```

만일 생성자 파라미터에 접근 제한자를 선언하지 않으면 생성자 파라미터는 생성자 내부에서만 유효한 지역 변수가 되어 생성자 외부에서 참조가 불가능하게 된다.

TYPESCRIPT

```
class Foo {
  // x는 생성자 내부에서만 유효한 지역 변수이다.
  constructor(x: string) {
    console.log(x);
  }
}

const foo = new Foo('Hello');
console.log(foo); // Foo {}
```

4. readonly 키워드

Typescript는 `readonly` 키워드를 사용할 수 있다. readonly가 선언된 클래스 프로퍼티는 선언 시 또는 생성자 내부에서만 값을 할당할 수 있다. 그 외의 경우에는 값을 할당할 수 없고 오직 읽기만 가능한 상태가 된다. 이를 이용하여 상수의 선언에 사용한다.

TYPESCRIPT

```
class Foo {
  private readonly MAX_LEN: number = 5;
  private readonly MSG: string;

  constructor() {
    this.MSG = 'hello';
  }

  log() {
    // readonly가 선언된 프로퍼티는 재할당이 금지된다.
    this.MAX_LEN = 10; // Cannot assign to 'MAX_LEN' because it is a constan
t or a read-only property.
    this.MSG = 'Hi'; // Cannot assign to 'MSG' because it is a constant or a
read-only property.

    console.log(`MAX_LEN: ${this.MAX_LEN}`); // MAX_LEN: 5
    console.log(`MSG: ${this.MSG}`); // MSG: hello
  }
}
```

```
new Foo().log();
```

5. static 키워드

ES6 클래스에서 static 키워드는 클래스의 정적(static) 메소드를 정의한다. 정적 메소드는 클래스의 인스턴스가 아닌 클래스 이름으로 호출한다. 따라서 클래스의 인스턴스를 생성하지 않아도 호출할 수 있다.

JAVASCRIPT

```
class Foo {
  constructor(prop) {
    this.prop = prop;
  }

  static staticMethod() {
    /*
    정적 메소드는 this를 사용할 수 없다.
    정적 메소드 내부에서 this는 클래스의 인스턴스가 아닌 클래스 자신을 가리킨다.
    */
    return 'staticMethod';
  }

  prototypeMethod() {
    return this.prop;
  }
}

// 정적 메소드는 클래스 이름으로 호출한다.
console.log(Foo.staticMethod());

const foo = new Foo(123);
// 정적 메소드는 인스턴스로 호출할 수 없다.
console.log(foo.staticMethod()); // Uncaught TypeError: foo.staticMethod is
 not a function
```

Typescript**에서는** static **키워드를 클래스 프로퍼티에도 사용할 수 있다**. 정적 메소드와 마찬가지로 정적 클래스 프로퍼티는 인스턴스가 아닌 클래스 이름으로 호출하며 클래스의 인스턴스를 생성하지 않아도 호출할 수 있다.

TYPESCRIPT

```
class Foo {
  // 생성된 인스턴스의 갯수
  static instanceCounter = 0;
  constructor() {
    // 생성자가 호출될 때마다 카운터를 1씩 증가시킨다.
    Foo.instanceCounter++;
  }
}

var foo1 = new Foo();
var foo2 = new Foo();

console.log(Foo.instanceCounter);  // 2
console.log(foo2.instanceCounter); // error TS2339: Property 'instanceCounte
r' does not exist on type 'Foo'.
```

6. 추상 클래스

추상 클래스(abstract class)는 하나 이상의 추상 메소드를 포함하며 일반 메소드도 포함할 수 있다. 추상 메소드는 내용이 없이 메소드 이름과 타입만이 선언된 메소드를 말하며 선언할 때 abstract 키워드를 사용한다. 추상 클래스를 정의할 때는 abstract 키워드를 사용하며, 직접 인스턴스를 생성할 수 없고 상속만을 위해 사용된다. 추상 클래스를 상속한 클래스는 추상 클래스의 추상 메소드를 반드시 구현하여야 한다.

TYPESCRIPT

```
abstract class Animal {
  // 추상 메소드
  abstract makeSound(): void;
  // 일반 메소드
  move(): void {
    console.log('roaming the earth ... ');
```

```
  }
}

// 직접 인스턴스를 생성할 수 없다.
// new Animal();
// error TS2511: Cannot create an instance of the abstract class 'Animal'.

class Dog extends Animal {
  // 추상 클래스를 상속한 클래스는 추상 클래스의 추상 메소드를 반드시 구현하여야 한다
  makeSound() {
    console.log('bowwow~~');
  }
}

const myDog = new Dog();
myDog.makeSound();
myDog.move();
```

인터페이스는 모든 메소드가 추상 메소드이지만 추상 클래스는 하나 이상의 추상 메소드와 일반 메소드를 포함할 수 있다.

Reference

- ECMAScript6 - Class
- Typescript Class
- Typescript Interface