[TS] 2. Class in TypeScript

by. Jbee

TypeScript에서의 클래스(class)에 대해 다뤄보겠습니다.

Table of contents

자바스크립트 개발자라고 하더라도 ES6덕분에 우리는 클래스에 이미 많이 익숙해져 있습니다. 그러나 ES6의 클래스는 뭔가가 조금 아쉬웠는데요, 이 부분을 TypeScript가 채워줄 예정입니다. 일단 ES6에서 클래스를 살펴봅니다.

ES6 code

```
class Person {
  constructor(firstName, lastName) {
    this._firstName = firstName;
    this._lastName = lastName;
  }

  get fullName() {
    return `${this._firstName} ${this._lastName}`;
  }
}

const p1: Person = new Person("Jbee", "Han");
console.log(p1.fullName); // Jbee Han
```

익숙한 클래스의 형태라고 생각합니다. 익숙하지 않으시다면 ES6 Class에 대한 포스팅을 참고해주세요. TypeScript에서의 클래스에는 ES6 코드와 비교해봤을 때, 이것 저것 키워드가 많이 추가가 되는데요, 하나씩 알아보기로 합시다.

1. Constructor

```
class Person {
  name: string;

  constructor(name, gender) {
    // ...
    this.gender = gender; // Error!
  }
}
```

ES6에서는 constructor로 받은 인자를 해당 클래스에 멤버로 등록하기 위해 this로 등록을 해줬습니다. 그 제한 또한 없었습니다. 하지만 타입스크립트에서는 해당 클래스의 프로퍼티로 등록되지 않은 속성에 대해 constructor에서 받을 수 없습니다. 위 코드에서는 `gender` 라는 프로퍼티를 정의하지 않았기 때문에 constructor에서 인자로 받을 수 없습니다.

> 프로퍼티(or filed member)로 미리 정의되지 않은 인자를 constructor에서 받을 수 없습니다.

2. extend를 통한 상속

TypeScript에서도 ES6와 마찬가지 방식으로 상속을 구현할 수 있습니다. `public`, `protected` 접근제어자로 정의된 프로퍼티와 메소드에 접근할 수 있습니다. (접근제어자에 대해서는 바로 다음 section에서 알아봅니다.)

```
class Person {
  name: string;
  constructor(name: string) {
    this.name = name;
  }
  sayName() {
    console.log(this.name);
  }
}

class Developer extends Person {
  constructor(name: string) {
    super(name);
  }
}

const devPerson: Person = new Developer("jbee");
devPerson.sayName(); // jbee
```

`sayName`은 `Person`클래스에만 정의되어 있는 메소드입니다. `Person`에서 정의한 대로 `this.name`을 console에 출력하는 것을 확인할 수 있습니다.

> extends 키워드를 통해서 클래스를 상속받아 부모 클래스의 메소드와 프로퍼티에 접근할 수 있습니다.

super

자식 클래스에서 constructor를 정의하려면 `super()`를 꼭 호출해줘야 합니다. (이는 ES6 문법과 동일합니다.) `super()`로 넘어가게 되는 인자 또한 부모 클래스에서 정의한 signature와 동일해야 합니다. 물론 별도로 정의하지 않으면 부모 클래스의 constructor를 따라갑니다.

Override method

`Developer` 클래스에서 해당 메소드 구현하게 되면 부모 클래스의 메소드를 override 하게 됩니다.

```
class Developer extends Person {
  constructor(name: string) {
    super(name);
  }

  sayName() {
    cosole.log(`I'm developer, ${this.name}`);
  }
}

const devPerson: Developer = new Developer("jbee");
devPerson.sayName(); // I'm developer, jbee
```

`sayName`이라는 메소드는 `Developer`클래스에서 재정의된 형태로 console에 출력합니다.

> 부모 클래스를 override하게 되면 부모 클래스에서 정의된 메소드를 재정의할 수 있습니다.

한 가지 짚고 넘어가자면, 자바에서 지원되는 overloading이 지원되지 않습니다. 따라서 다음과 같은 메소드는 추가할 수 없습니다.

```
class Developer extends Person {
  // ...
  sayName(position) { // Error!
    // ...
  }
}
```

`[!] Types of property 'sayName' are incompatible.` 라는 에러가 발생합니다. 자식 클래스에서 부모 클래스와 메소드의 이름은 동일하고 signature가 다른 메소드는 정의할 수 없습니다. 이미 부모 클래스에 `sayName` 이라는 메소드가 존재하기 때문에 해당 메소드와 동일한 signature로 override하지 않는 이상, 다른 signature로 또다른 메소드를 정의할 수 없습니다.

> [!] 하지만 `any` 라는 타입과 함께 오버로딩을 메소드 내부에서 if 문 또는 switch 문으로 분기하여 비슷하게 구현할 수 있습니다. **바로 다음에 다룰** Function **포스팅에서 다루겠습니다**.

3. Access Modifier

TypeScript에서는 클래스의 프로퍼티 또는 메소드에 접근제어자를 추가할 수 있습니다.

public (by default)

타입스크립트에서 아무 접근 제어자를 추가하지 않으면 기본적으로 `public` 접근제어자와 동일하게 동작합니다. ES6에서와 마찬가지로 클래스 내부, 외부에서 모두 접근할 수 있는 프로퍼티를 정의할 때 사용합니다. `getter`, `setter`를 별도로 만들지 않아도 접근이 가능합니다. 위에서 정의한 `Person` 클래스를 사용하여 예시를 보겠습니다.

```
const p1 = new Person("jbee");
console.log(p1.name); // "jbee"
```

프로퍼티에 바로 접근할 수 있는 것을 확인할 수 있습니다.

private

`private` 접근 제어자로 정의된 멤버는 클래스 밖에서 접근할 수 없습니다. 이번 예시 코드에서는 `getter`, `setter`를 추가해주기 위해 프로퍼티 명 앞에 `_`(언더바)를 추가했습니다.

```
class Person {
  name: string;
  private _job: string;

  constructor(name, job) {
    // ...
  }
}

const p1: Person = new Person("jbee", "Developer");
console.log(p1._job); // Error!
```

`Property '_job' is private and only accessible within class 'Person'.` 이라는 에러가 발생합니다. 클래스 내부가 아닌 외부에서 `private` 으로 정의된 프로퍼티에 접근하려고 했기 때문에 발생하는 에러입니다. 해당 프로퍼티에 접근하기 위해 `getter` 를 추가하겠습니다.

```
class Person {
  // ...
  get job() {
    return this._job;
  }
}
const p1: Person = new Person("jbee", "Developer");
console.log(p1.job); // Developer
```

이렇게 접근할 수 있게 됩니다. `getter`, `setter`를 추가하는 것보다 접근 제어자를 `public`하는 것이 맞는 것 같습니다.

> 프로퍼티 또는 메소드에 `private`이라는 접근 제한자를 붙이게 되면 클래스 내에서만 사용할 수 있는 프로퍼티, 그리고 메소드가 됩니다.

protected

기본적으로는 `private` 접근 제어자와 동일하게 동작합니다. 하지만 어느 한 곳에서는 접근이 가능한데요, 바로 해당 클래스를 상속한 클래스에서 접근이 가능합니다.

```
class Perosn {
  protected isWorking: boolean;
  // ...
}

const p1: Person = new Person(true);
console.log(p1.isWorking);// Error!
```

`[!] Property 'isWorking' is protected and only accessible within class 'Person' and its subclasses.` 라는 에러가 발생합니다. `private` 접근제어자와 마찬가지로 클래스 외부에서 접근할 수 없음을 뜻합니다.

```
class Developer extends Person {
  constructor(isWorking) {
    super(isWorking);
  }

  isWork() {
    console.log(this.isWorking);
  }
}

const devPerson: Developer = new Developer(true);
devPerson.isWork(); // true
```

위 `Person` 클래스를 상속한 클래스에서는 `protected` 로 정의된 프로퍼티에 접근할 수 있습니다.

> 프로퍼티 또는 메소드에 `protected` 이라는 접근 제한자를 붙이게 되면 클래스 내에서와 해당 클래스를 상속한 클래스 안에서만 사용할 수 있게 됩니다.

이 접근 제어자는 `constructor` 에도 추가될 수 있습니다.
`constructor` 에 해당 접근제어자를 추가하는 경우, 해당 클래스는 바로 인스턴스화될 수 없으며 이 클래스를 상속받은 클래스에서 `super` 라는 키워드로 호출이 가능합니다.

```
class Perosn {
  protected constructor() {
    // ...
  }
}
class Developer extends Person {
  constructor() {
    super();
  }
}

const p1: Person = new Person(true); // Error!
const devPerson: Developer = new Developer(); // OK!
```

`[!] Constructor of class 'Person' is protected and only accessible within the class declaration.` 라는 에러가 발생하며 상속받은 클래스만 인스턴스화 시킬 수 있습니다. 추상클래스(Abstract class)도 마찬가지로 상속을 하기 위한 클래스인데요, 추상 클래스는 구현되지 않은 메소드가 존재하는 반면 이 방식은 모든 메소드가 구현되어야 합니다.

ES.NEXT

tc39/proposal-class-field를 참고해보시면, ECMAScript에서도 `private`에 대한 스펙이 논의 중이며(stage-3) 해당 스펙은 `private`이라는 키워드 대신 `#`이라는 키워드로 스펙이 논의 중입니다.

readonly

자바를 공부하셨던 분이라면 `final` 키워드를 아실텐데요, TypeScript에서는 읽기 전용 프로퍼티를 설정하기 위해 `readonly` 프로퍼티를 사용합니다.

```
class Person {
  public readonly age: number;

  // ...

  set setAge(age: number) {
    this.age = age; // Error!
  }
}

const p1: Person = new Person(25);
p1.age = 20; // Error!
```

`[!] Cannot assign to 'age' because it is a constant or a read-only property.` 라는 에러가 발생합니다! `readonly` 키워드가 추가되면 상수(constant)로 인식하게 됩니다.

> `readonly` 와 함께 정의된 프로퍼티는 `constructor` 에서 한 번 결정되면 수정할 수 없습니다.

static

`static` 키워드는 ES6에서와 동일하게 사용되며 사용할 수 있습니다. 다만 ES6에서는 메소드에만 해당 키워드를 추가하는 것이 가능했는데요, TypeScript에서는 `static` 키워드를 프로퍼티에도 추가할 수 있습니다. 해당 키워드를 프로퍼티 또는 메소드에 추가하게 되면 인스턴스를 생성하지 않고 접근하거나 생성할 수 있습니다.

ES6 code

```
class Circle {
  // ...
}
Circle.center = [0,0]
```

TypeScript code

```
class Circle {
  static center: number[];
  //...
}
```

마무리

TypeScript에서 사용하는 클래스에 대해 알아봤습니다. 잘 사용하던 ES6에서의 클래스가 조금 덜떨어져 보일 수 있습니다!

감사합니다.