타입스크립트 #1

# 인터페이스

인터페이스란 단어를 처음 들어보신분들도 있고, 이름은 익숙하지만 제대로 이해하지 못한 사람들이 많을 것이라 예상합니다. 타입스크립트의 인터페이스 사용법을 설명하기 전 인터페이스란 어떤 것인지 알아보도록 하겠습니다.

인터페이스란 ? OOP에서 특정 자료형이 가져야할 메서드와 프로퍼티를 명시 해놓은 일종의 명세서입니다. 아직 감이 오지 않으리라 생각이 듭니다. 예를 들어 자바스크립트 개발자 A와 B가 각각 Dog클래스와 Cat클래스를 만드는 프로젝트를 진행하고 있습니다. 이들에게는 잘 짜놓은 팀 규칙이 존재하지만 개발자 B는 가끔 팀 규칙을 무시할 때가 많습니다. 그래서, Dog클래스와 Cat클래스는 {age : number, name : string, sex : string, fullness : number, eat : (food) => number} 의 프로퍼티와 메서드를 가져야 하지만, B개발자는 fullness 를 Boolean, eat : (food) => void 로 Cat클래스를 작성하고, sex를 cat의 프로퍼티로 지정하지 않았습니다. A개발자는 추후 Master라는 클래스를 만들어 Dog과 Cat의 인스턴스를 가지도록 하고, 각각의 Dog과 Cat의 메서드와 프로퍼티를 이용하도록 작성하였습니다. 그러나, 두 파일을 머지하고 보니, 빌어먹을 B개발자가 공통적으로 가져야하는 프로퍼티와 메서드를 자기 마음대로 짠 걸 확인 했습니다. 이후 B개발자는 A개발자에게 사과를 하고, B개발자는 하루를 버려가며, 다시 Cat 클래스를 다시 만들었습니다. 애초에 팀 규칙을 프로그래밍적으로 강제 했으면, 두 파일을 머지하기 전에 Cat클래스를 제대로 만들고 하루를 날리는 일은 없었을 텐데 말이죠.

인터페이스는 위와 같은 상황에서 클래스가 가져야 하는 프로퍼티와 메서드를 프로그래밍 단계에서 강제 할 수 있는 힘이 존재합니다. 다시 말해 클래스가 명세인 인터페이스를 따르지 않는 다면, 계속 에러를 발생 시켜 프로그래머에게 올바른 클래스의 작성을 명령 합니다.

# 인터페이스 사용

interface FoodInterface{

name : string,

calory : number

}

interface AnimalInterface{

name : string,

fullness : number,

eat : (food : FoodInterface) => number

}

i

인터페이스를 사용하기 위해서는 interface 예약어를 사용합니다. Interface또한 Class 식별자 작성법 처럼 첫 알파벳은 대문자를 사용하고, 클래스와 인터페이스명간의 혼돈을 피하고자 interface접미사를 붙여주는것이 일반적입니다.

그럼 강아지와 고양이를 객체 리터럴로 인스턴스를 만들어 인터페이스가 제대로 작동하는지 확인해보도록 하겠습니다.

// ...

const dog : AnimalInterface = {

name : "gogo",

fullness : 0,

eat : function(food : FoodInterface){

return this.fullness + food.calory

}

}

console.log(dog);

제대로 Interface대로 작성되어 있는 dog에서 fullness를 string타입으로 한번 바꿔 보겠습니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Tsc는 dog인스턴스의 fullness가 string 타입이 아닌 number타입이며, animalInterface대로 작성하라고 요구 하고 있습니다. 이처럼 Interface는 어떤 인스턴스 혹은 자료형에 프로퍼티와 메서드를 강제할 수 있는 힘이 존재합니다. 이번에는 인스턴스가 아닌 클래스를 작성할 때 인터페이스를 사용해보도록 하겠습니다.

interface AnimalInterface{

name : string,

fullness : number,

eat : (food : FoodInterface) => number

}

//

class Dog implements AnimalInterface{

name : string;

fullness : number;

eat : (food : FoodInterface) => number;

constructor(){

this.name = "gogo";

this.fullness = 0;

this.eat = function(food : FoodInterface){

return this.fullness+= food.calory;

}

}

}

여기 까지 작성 했다면 아무 문제 없이 interface.js파일로 트랜스파일링 될 것 입니다. 이제 interface와 다르게 eat부분을 모두 주석 처리 해보도록 하겠습니다.

//...

class Dog implements AnimalInterface{

name : string;

fullness : number;

// eat : (food : FoodInterface) => number;

constructor(){

this.name = "gogo";

this.fullness = 0;

// this.eat = function(food : FoodInterface){

// return this.fullness+= food.calory;

// }

}

}

이제 다시 트랜스파일링을 시도하니, AnimalInterface와 다르게 작성되어 있다고 에러를 리턴 합니다. 다시 주석되어 있는 부분을 원상태로 변경하고, 이번에는 interface를 이용하여, Dog의 인스턴스를 생성 할 때 생성자로 이름을 받아 this.name에 할당하도록 강제하겠습니다.

# Interface 생성자 및 확장

AnimalInterface아래에 새로운 interface AnimalConstructor를 다음과 같이 작성해줍니다.

interface AnimalInterface{

name : string,

fullness : number,

eat : (food : FoodInterface) => number

}

interface AnimalConstructor{

new (name : string) : AnimalInterface;

}

생성자를 interface에서 사용 할 때는 new 키워드를 사용하며, 이 생성자로 만들어진 인스턴스가 어떤 타입인지를 작성해주어야 합니다. 그리고는 다음과 같이 작성합니다.

//...

interface AnimalInterface{

name : string,

fullness : number,

eat : (food : FoodInterface) => number

}

interface AnimalConstructor{

new (name : string) : AnimalInterface;

}

const Dog : AnimalConstructor = class Dog implements AnimalInterface{

name : string;

fullness : number;

eat : (food : FoodInterface) => number;

constructor(name : string){

this.name = name;

this.fullness = 0;

this.eat = function(food : FoodInterface){

return this.fullness+= food.calory;

}

}

}

const myFirstDog = new Dog("gogo");

이런 의문이 들겁니다. 왜 class Dog implements AnimalConstructor가 아니라 AnimalInterface이며, 이 클래스를 Dog 변수에 할당하며, 이 클래스의 자료형은 AnimalConstructor로 되어 있는지요.

이를 위해서는 스태틱과 인스턴스를 구분 하여야합니다. Implements는 인스턴스를 검사하여, 해당 인스턴스가 interface를 따라가는지를 구분 합니다. 그러나, static 메서드인 constructor는 이 검사에서 빠지게 되어, 생성자 시그네쳐를 따라가게 된다면 에러를 발생 시키게 됩니다. 그렇기에, const Dog에 생성자 시그네쳐가 존재하는 interface를 부여하게 된다면, 이 변수는 class의 생성자를 리턴하게 되기에 문제 없이 작성 할 수 있습니다.

이제 AnimalInterface를 가지고 있지만 구분되는 Bird Class를 작성해보도록 하겠습니다. Bird클래스는 fly메서드를 가지도록 강제 하겠습니다. 먼저 Bird인터페이스와 Bird클래스를 작성하도록 하겠습니다.

//...

interface BirdInterface{

fly : () => void

}

class Bird implements BirdInterface{

fly : ()=> void;

constructor(){

this.fly = function(){

console.log("난다 !!!")

}

}

}

만약 여기서 BirdInterface에 AnimalInterface를 따라하기 위해 AnimalInterface 내용을 또 적어주기는 고통스러울 겁니다. 만약 이처럼 확장되어야 하는 Interface가 많다면 코드상에 겹치는 내용은 많고 가시성은 떨어지게 될 겁니다. 이때 확장의 개념을 도입하여, Interface의 활용을 증가시키도록 하겠습니다.

인터페이스를 확장하는 방법은 extends키워드를 사용하는 겁니다. BirdInterface에 AnimalInterface를 확장 시켜, AnimalInterface의 명세를 잇도록 합니다. 그럼 Bird클래스는 AnimalInterface의 명세를 지켜달라고 요구합니다. 그럼 Dog클래스의 내용을 복사해서 Bird에도 적용시키도록 해봅시다. 어떤가요. Bird클래스도 AnimalInterface가 요구하는 대로 작성하게 되었습니다.

//...

interface BirdInterface extends AnimalInterface{

fly : () => void

}

class Bird implements BirdInterface{

name : string;

fullness : number;

eat : (food : FoodInterface) => number;

fly : ()=> void;

constructor(name : string){

this.fly = function(){

console.log("난다 !!!")

}

this.name = name;

this.fullness = 0;

this.eat = function(food : FoodInterface){

return this.fullness+= food.calory;

}

}

}

이제 AnimalInterface를 따르는 팩토리 펑션으로 리팩토링 해보도록 하겠습니다. 먼저 const Dog 을 삭제하도록 합니다. 그리고 createAnimal함수를 정의 합니다.

function createAnimal(AnimalClass : AnimalConstructor,name : string) : AnimalInterface{

return new AnimalClass(name);

}

이 함수는 AnimalInterface를 가진 인스턴스를하며, AnimalConstructor를 가지고 있는 생성자를 매개변수로 받아 이 함수로는 AnimalInterface를 따르는 class만 생성 할 수 있습니다. 그래서 문서 아래에 다음과 같이 이 createAnimal 함수로 저희의 첫 새와 강아지를 생성 해보도록 하겠습니다.

const myFirstDog = createAnimal(Dog, "gogo");

const myFirstBird = createAnimal(Bird, "dodo");

console.log(myFirstBird);

console.log(myFirstDog)

문제 없이 트랜스 파일링 되었고, 문제 없이 js파일에서도 인스턴스가 생성된 것을 확인 할 수 있습니다.