인터페이스

Introduction

인터페이스는 일반적으로 **타입 체크를 위해 사용되며 변수, 함수, 클래스에 사용**할 수 있다. 인터페이스는 여러가지 타입을 갖는 프로퍼티로 이루어진 새로운 타입을 정의하는 것과 유사하다. 인터페이스에 선언된 프로퍼티 또는 메소드의 구현을 강제하여 일관성을 유지할 수 있도록 하는 것이다. ES6는 인터페이스를 지원하지 않지만 TypeScript는 인터페이스를 지원한다.

인터페이스는 프로퍼티와 메소드를 가질 수 있다는 점에서 클래스와 유사하나 직접 인스턴스를 생성할 수 없고 모든 메소드는 추상 메소드이다. 단, 추상 클래스의 추상 메소드와 달리 abstract 키워드를 사용하지 않는다.

변수와 인터페이스

인터페이스는 변수의 타입으로 사용할 수 있다. 이때 인터페이스를 타입으로 선언한 변수는 해당 인터페이스를 준수하여야 한다. 이것은 새로운 타입을 정의하는 것과 유사하다.

```
// 인터페이스의 정의
interface Todo {
  id: number;
  content: string;
  completed: boolean;
}

// 변수 todo의 타입으로 Todo 인터페이스를 선언하였다.
let todo: Todo;

// 변수 todo는 Todo 인터페이스를 준수하여야 한다.
todo = { id: 1, content: 'typescript', completed: false };
```

인터페이스를 사용하여 함수 파라미터의 타입을 선언할 수 있다. 이때 해당 함수에는 함수 파라미터의 타입으로 지정한 인터페이스를 준수하는 인수를 전달하여야 한다. 함수에 객체를 전달할 때 복잡한 매개변수 체크가 필요없어서 매우 유용하다.

```
// 인터페이스의 정의
interface Todo {
  id: number;
  content: string;
  completed: boolean;
}

let todos: Todo[] = [];

// 파라미터 todo의 타입으로 Todo 인터페이스를 선언하였다.
function addTodo(todo: Todo) {
  todos = [...todos, todo];
}

// 파라미터 todo는 Todo 인터페이스를 준수하여야 한다.
const newTodo: Todo = { id: 1, content: 'typescript', compl
eted: false };
addTodo(newTodo);
console.log(todos)
// [ { id: 1, content: 'typescript', completed: false } ]
```

함수와 인터페이스

인터페이스는 함수의 타입으로 사용할 수 있다. 이때 함수의 인터페이스에는 타입이 선언된 파라미터 리스트와 리턴 타입을 정의한다. 함수 인테페이스를 구현하는 함수는 인터페이스를 준수하여야 한다.

```
// 함수 인터페이스의 정의
interface SquareFunc {
  (num: number): number;
}

// 함수 인테페이스를 구현하는 함수는 인터페이스를 준수하여야 한다.
```

```
const squareFunc: SquareFunc = function (num: number) {
  return num * num;
}

console.log(squareFunc(10)); // 100
```

클래스와 인터페이스

클래스 선언문의 implements 뒤에 인터페이스를 선언하면 해당 클래스는 지정된 인터페이스를 반드시 구현하여야 한다. 이는 인터페이스를 구현하는 클래스의 일관성을 유지할 수 있는 장점을 갖는다. 인터페이스는 프로퍼티와 메소드를 가질 수 있다는 점에서 클래스와 유사하나 직접 인스턴스를 생성할 수는 없다.

```
// 인터페이스의 정의
interface ITodo {
  id: number;
  content: string;
  completed: boolean;
}

// Todo 클래스는 ITodo 인터페이스를 구현하여야 한다.
class Todo implements ITodo {
  constructor (
    public id: number,
    public content: string,
    public completed: boolean
  ) { }
}

const todo = new Todo(1, 'Typescript', false);

console.log(todo);
```

인터페이스는 프로퍼티뿐만 아니라 메소드도 포함할 수 있다. 단, 모든 메소드는 추상 메소드이어야 한다. 인터페이스를 구현하는 클래스는 인터페이스에서 정의한 프로퍼티와 추상

메소드를 반드시 구현하여야 한다.

```
// 인터페이스의 정의
interface IPerson {
  name: string;
  sayHello(): void;
}

/*
인터페이스를 구현하는 클래스는 인터페이스에서 정의한 프로퍼티와 추상 메소드를 반드시 구현하여야 한다.
*/
class Person implements IPerson {
  // 인터페이스에서 정의한 프로퍼티의 구현
  constructor(public name: string) {}

  // 인터페이스에서 정의한 추상 메소드의 구현
  sayHello() {
    console.log(`Hello ${this.name}`);
  }
}

function greeter(person: IPerson): void {
  person.sayHello();
}

const me = new Person('Lee');
greeter(me); // Hello Lee
```

덕 타이핑 (Duck typing)

주의해야 할 것은 인터페이스를 구현하였다는 것만이 타입 체크를 통과하는 유일한 방법은 아니다. 타입 체크에서 중요한 것은 값을 실제로 가지고 있는 것이다. 이해가 어려울 수 있으므로 예를 들어 설명한다.

```
interface IDuck { // 1
  quack(): void;
```

```
}

class MallardDuck implements IDuck { // 3
  quack() {
    console.log('Quack!');
  }
}

class RedheadDuck { // 4
  quack() {
    console.log('q~uack!');
  }
}

function makeNoise(duck: IDuck): void { // 2
  duck.quack();
}

makeNoise(new MallardDuck()); // Quack!
makeNoise(new RedheadDuck()); // q~uack! // 5
```

(1) 인터페이스 IDuck은 quack 메소드를 정의하였다.

(2) makeNoise 함수는 인터페이스 IDuck을 구현한 클래스의 인스턴스 duck을 인자로 전달받는다.

(3) 클래스 MallardDuck은 인터페이스 IDuck을 구현하였다.

(4) 클래스 RedheadDuck은 인터페이스 IDuck을 구현하지는 않았지만 quack 메소드를 갖는다.

(5) makeNoise 함수에 인터페이스 IDuck을 구현하지 않은 클래스 RedheadDuck의 인스턴스를 인자로 전달하여도 에러 없이 처리된다.

TypeScript는 해당 인터페이스에서 정의한 프로퍼티나 메소드를 가지고 있다면 그 인터페이스를 구현한 것으로 인정한다. 이것을 덕 타이핑(duck typing) 또는 구조적 타이핑(structural typing)이라 한다.

인터페이스를 변수에 사용할 경우에도 덕 타이핑은 적용된다.

```
interface IPerson {
  name: string;
}

function sayHello(person: IPerson): void {
  console.log(`Hello ${person.name}`);
}

const me = { name: 'Lee', age: 18 };
sayHello(me); // Hello Lee
```

변수 me는 인터페이스 IPerson과 일치하지는 않는다. 하지만 IPerson의 name 프로퍼티를 가지고 있으면 인터페이스에 부합하는 것으로 인정된다.

인터페이스는 개발 단계에서 도움을 주기 위해 제공되는 기능으로 자바스크립트의 표준이 아니다. 따라서 위 예제의 TypeScript 파일을 자바스크립트 파일로 트랜스파일링하면 아래와 같이 인터페이스가 삭제된다.

```
function sayHello(person) {
  console.log("Hello " + person.name);
}
var me = { name: 'Lee', age: 18 };
sayHello(me); // Hello Lee
```

선택적 프로퍼티

인터페이스의 프로퍼티는 반드시 구현되어야 한다. 하지만 인터페이스의 프로퍼티가 선택적으로 필요한 경우가 있을 수 있다. 선택적 프로퍼티(Optional Property)는 프로퍼티명 뒤에 `?`를 붙이며 생략하여도 에러가 발생하지 않는다.

```
interface UserInfo {
  username: string;
  password: string;
  age?    : number;
  address?: string;
```

```
}

const userInfo: UserInfo = {
  username: 'ungmo2@gmail.com',
  password: '123456'
}

console.log(userInfo);
```

이렇게 선택적 프로퍼티를 사용하면 사용 가능한 프로퍼티를 파악할 수 있어서 코드를 이해하기 쉬워진다.

인터페이스 상속

인터페이스는 extends 키워드를 사용하여 인터페이스 또는 클래스를 상속받을 수 있다.

```
interface Person {
  name: string;
  age?: number;
}

interface Student extends Person {
  grade: number;
}

const student: Student =  {
  name: 'Lee',
  age: 20,
  grade: 3
}
```

복수의 인터페이스를 상속받을 수도 있다.

```
interface Person {
  name: string;
  age?: number;
```

```
}

interface Developer {
  skills: string[];
}

interface WebDeveloper extends Person, Developer {}

const webDeveloper: WebDeveloper =  {
  name: 'Lee',
  age: 20,
  skills: ['HTML', 'CSS', 'JavaScript']
}
```

인터페이스는 인터페이스 뿐만 아니라 클래스도 상속받을 수 있다. 단, 클래스의 모든 멤버(public, protected, private)가 상속되지만 구현까지 상속하지는 않는다.

```
class Person {
  constructor(public name: string, public age: number) {}
}

interface Developer extends Person {
  skills: string[];
}

const developer: Developer =  {
  name: 'Lee',
  age: 20,
  skills: ['HTML', 'CSS', 'JavaScript']
}
```