Class and Interfaces

▼ Classes_abstract class와 protected

▼ abstract class

- 다른 클래스가 상속받을 수 있는 클래스 직접 새로운 인스턴스를 만들 수 없음
- 추상 클래스 안에서는 추상 메소드를 만들 수 있지만 메소드를 구현하여서는 안됨 대신 call signature 만 적어두어야 함

→ 추상 메소드는 추상 클래스를 상속받는 모든 것들을 구현해야하는 메소드를 의미
→ 상속받은 곳에서 구현이 어떻게 돼도 상관은 없지만, 구현은 해야함

- protected : 상속받은 자식에서 접근 가능
- 추상 메소드 property를 private로 만든다면, 클래스를 상속 하더라도 안의 변수를 사용할 수 없다.

```
abstract class User {
    constructor(
        protected firstName:string,
        private lastName:string,
        public nickname:string
    ){}

    abstract getNickName():void  //추상 메소드

    getFullName(){
        //implementation(구현)
        return `${this.firstName} ${this.lastName}` //firstName과 lastName을 합친 string을 리턴
    }
}


//일반 클래스
class Player extends User {
    //추상메소드  abstract getNickName()를 작성함으로 Player에서는 getNickName을 필수로 구현
    getNickName () {
        // console.log(this.lastName())   //private이기 때문에 lastName을 호출할 수 없음
        console.log(this.firstName)
```

```
    }
  }

  // const nico = new User("nico", "las", "니꼬") ⇒ 추상 클래스의 인스턴스를 만들 수 없음
  const nico = new Player("nico", "las", "니꼬")

  // nico.firstName   //private라서 오류 발생
  nico.nickname       //public이라서 오류 발생 X
  nico.getFullName()
```

▼ HashMap

```
//HashMap
type Words = {
   [key:string]: string
}

class Dict{
   private words: Words    //property가 constructor부터 바로 초기화 안됨
   constructor () {
      this.words = {}     //수동으로 초기화
   }
   add(word:Word){         //클래스를 타입처럼 쓸 수 있음
      if(this.words[word.term] === undefined){
         this.words[word.term] = word.def;
      }
   }
   del(word:Word){
      if(this.words[word.term] !== undefined){
         delete this.words[word.term]
      }
   }
   def(term:string){
      return this.words[term]
   }
}

class Word {
   constructor(
      public term:string,
      public def:string
   ) {}
}

const kimchi = new Word("kimchi", "한국의 음식")
const test = new Word("test", "이건 테스트")
```

```
const dict = new Dict()

dict.add(kimchi)
dict.add(test)
dict.def("kimchi")
```

▼ readonly, static

```
//Interfaces
type Words = {
    [key:string]: string
}

class Dict{
    private words: Words    //property가 constructor부터 바로 초기화 안됨
    constructor () {
        this.words = {}     //수동으로 초기화
    }
        ...
    //static 메서드
    static hello () {
        return "hello"
    }
}

class Word {
    constructor(
        //안의 속성이 public이기 때문에 수정이 가능하다.
        // public term:string,
        // public def:string

        public readonly term:string,
        public readonly def:string
    ) {}
}

const kimchi = new Word("kimchi", "한국의 음식")
kimchi.def = "변경할 수 없음"    //readonly로 인해 수정할 수 없음

Dict.hello()    //static 메서드는 정상작동
```

▼ Interfaces_1

▼ 타입스크립트에서 object의 모양을 알려주는 방법

- type 사용
- interface 사용

```
type Player = {
    nickname :string,
    team:Team,
    health?:Health
}
```

```
interface Player {
    nickname :string,
    team:Team,
    health?:Health
}
```

▼ Interfaces_01

```
//기본 예제
type Nickname = string
type Health = number
type Friends = Array<string>


type Player = {
    nickname:Nickname,
    healthBar:Health
}

const nico = {
    nickname:"nick",
    healthBar:10
}

type Food = string;
const kimchi:Food = "delicious"



//예제2: 타입을 지정된 옵션으로만 제한
type Team = "red" | "blue" | "yellow"   //concrete타입의 특정 값 사용 가능
type Health = 1 | 5 | 10


//type사용
type Player = {
    nickname :string,
    team:Team,
    health?:Health
}

//인터페이스 사용
//오로지 오브젝트의 모양을 특정하는 용도
interface Player {
    nickname :string,
    team:Team,
```

```
    health?:Health
}

const nico:Player = {
    nickname:"nico",
    // team:"pink"      //허용되지 않는 값
    team : "blue",
    health: 10
}
```

▼ interface_02

- 인터페이스는 각각 생성해도 ts가 알아서 하나로 만들어줌 ⇒ type은 안됨

```
interface User {
    name:string
}

interface User {
    lastName:string
}

interface User {
    health:number
}

const nico:User = {
    name:"nico",
    lastName:"n",
    health:10
}
```

▼ Interfaces_2

```
////Interface2
interface User{
    firstName:string,
    lastName:string,
    sayHi(name:string):string
    fullName():string
}

interface Human{
    health:number
}
```

```
class Player implements User, Human {
    constructor(
        public firstName:string,
        public lastName:string,
        public health:number
    ){}
    fullName(){
        return `${this.firstName} ${this.lastName}`
    }
    sayHi(name:string){
        return `Hello ${name}. My name is ${this.fullName()}`
    }
}

//이렇게 사용할 필요가 없다.
// function makeUser (user:User): User{
//     new Player()...
// }

function makeUser (user:User): User{
    return{
        firstName:"nico",
        lastName:"las",
        fullName:() => "xx",
        sayHi:(name) => "string"
    }
}

//인자를 써서 객체의 모양을 지정해 줄 수 있다
makeUser({
    firstName:"nico",
    lastName:"las",
    fullName:() => "xx",
    sayHi:(name) => "string"
})
```

▼ interface_정리

▼ type vs interface

- type : 추후 속성을 추가하고 싶을 때, 덮어쓰기를 할 수 없다.
- interface : type과 달리 interface는 같은 이름으로 덮어쓰기를 할 수 있다.
- **즉, ts에서는 객체의 모양을 알려주기 위해서는 인터페이스를 사용하고, 나머지 상황에서는 타입을 사용한다.**

```
///type사용
type PlayerA = {
```

```
    name:string
}

//상속
type PlayerAA = PlayerA & {
    lastName:string
}

//추후 속성을 추가하고 싶을 때, 덮어쓰기를 할 수 없다.
// type PlayerAA = PlayerA & {
//     health:number
// }

const playerA: PlayerAA = {
    name:"nico",
    lastName:"las"
}


///interface사용
interface PlayerB {
    name:string
}

//상속
interface PlayerBB extends PlayerB {
    lastName:string
}

//type과 달리 interface는 같은 이름으로 덮어쓰기를 할 수 있다.
interface PlayerBB {
    health:number
}

const playerB: PlayerBB = {
    name:"nico",
    lastName:"xxx"
}
```

▼ type과 interface 둘다 상속 받을 수 있다.

```
type PlayerA = {
    firstName:string
}

interface PlayerB {
    firstName:string
```

```
}

class User implements PlayerA {
    constructor(
        public firstName:string
    ){}
}

class User implements PlayerB {
    constructor(
        public firstName:string
    ){}
}
```

▼ Polymorphism

```
// Storage interface ⇛ 이미 선언된 자바스크립트의 웹 스토리지 API를 위한 인터페이스
interface SStorage<T> {
    [key:string]: T
}

class LocalStorage<T> {
    private storage: SStorage<T> = {

    }

    set(key:string, value:T){
        this.storage[key] = value;
    }

    remove(key:string){
        delete this.storage[key]
    }

    get(key:string):T {
        return this.storage[key]
    }

    clear(){
        this.storage ={

        }
    }
}
```

```
const stringStorage = new LocalStorage<string>()

stringStorage.get("ket")
stringStorage.set("hello", "hello how are you")

const booleanStorage = new LocalStorage<boolean>();
booleanStorage.get("xxx");  //받을 때 boolean으로 받음
booleanStorage.set("hello", true)   //두번째 인자가 자동으로 true가 되어야 함
```

▼ 정리

- interface vs type

  Type의 용도 :
  1. 특정 값이나 객체의 값에 대한 타입을 지정해줄 수 있다.
  2. Type alias(타입에 대한 별명)를 만들어줄 수 있다.
  3. 타입을 특정한 값을 가지도록 제한할 수 있다.

  타입과 인터페이스의 차이점은 type 키워드는 interface 키워드에 비해서 좀 더 활용할 수 있는 것이 많다

  즉, interface는 오로지 객체의 형태를 타입스크립트에게 설명해주기 위한 용도로만 사용된다 !

  interface는 위와 같이 상속의 개념을 사용할 수 있다 ! (오른쪽은 type을 이용하여 상속한 방법)
  ⇒ 문법 구조가 객체지향 프로그래밍의 개념을 활용 ⭐

  인터페이스의 또 다른 특징으로는 속성(Property)들을 '축적'시킬 수 있다는 것이다.

- 인터페이스 사용

  만약 추상 클래스를 다른 클래스들이 특정 모양을 따르도록 하기 위한 용도로 사용할 경우
  같은 역할을 하는 인터페이스를 사용하는 것이 좋음
  class Player extends User ⇒ class Player implement User