클래스와 인터페이스

1. 클래스와 상속

클래스를 선언하는 방법 : class 키워드를 사용한다.

```
class A {...}
```

다른 클래스를 상속하는 방법 : extends 키워드를 이용한다.

```
class A extends B {...}
```

자바스크립트에서 Class를 선언해주는 방법은 아래와 같다.

```
class Car {
        constructor(color){
                this.color = color
        }
        start(){
                        console.log('start');
        }
}
```

하지만 타입스크립트에서 위와 같이 작성하면 에러가 난다.

TS에서 멤버변수는 미리선언해줘야한다. (미리 선언하지 않아도 readonly나 접근제한자를 사용할 수도 있다.)

```
class Car {
        color:string
        constructor(color:string){
                this.color = color
        }
        start(){
                        console.log('start');
        }
}
// 혹은
class Car {
        constructor(readonly color:string){
                this.color = color
        }
        start(){
                        console.log('start');
        }
}
```

abstract 추상클래스

class 키워드 앞에 abstract 키워드를 붙이면 추상 클래스를 사용할 수 있다.

추상 클래스의 특징은 다음과 같다.

- 인스턴스를 생성할 수 없다.

```
abstract class A { ... }
new A(); //Error
```

- 상속을 받은 클래스를 통해서만 인스턴스화 할수 있다.

```
abstract class A { ... }

class B extends A { ... }

new B() // OK
```

- abstract 클래스 내부에 선언한 abstract 메서드는 상속 받은 클래스에서 무조건 선언해줘야한다.

```
abstract class A {
    // ...
    moveTo(){...}
    abstract canMoveTo(): boolean
}
```

접근 한정자

메서드, 클래스의 인스턴스 프로퍼티, 생성자의 매개변수나 프로퍼티 초기자에 접근 한정자를 사용하여 외부/내부의 접근을 통제할 수 있다.

- public : 어디에서나 접근 가능하며 기본적으로 주어지는 접근 수준이다.
- private(#): 해당 클래스의 내부에서만 접근 할 수 있다.
- protected: 해당 클래스와 자식 클래스의 인스턴스에서만 접근 가능하다.

2. Super

super 키워드란 자식 인스턴스에서 부모 인스턴스의 메서드에 접근 할 수 있는 키워드이다.

TS에서는 두가지의 super 호출을 지원한다.

1. super.take 과 같은 메서드 호출
2. 자식과 부모 클래스 모두 생성자 함수가 존재한다면, super()메서드로 부모 클래스의 생성자를 호출해야 한다.

super로 부모 클래스의 메서드에만 접근할 수 있고 프로퍼티에는 접근할 수 없다.

3. 반환 타입으로서의 this

클래스를 정의할 때 메서드의 반환 타입으로 this타입을 사용할 수 있다.

this를 사용하지 않고 해당 클래스를 return 하는 함수를 만들때 그 부모를 상속하는 모든 자식 클래스는 다시 오버라이드 해야한다.

this 키워드를 사용한다면, 반환타입으로 해당 인스턴스를 리턴해준다.

this 키워드를 사용하지 않는 다면, Set을 상속받는 클래스에 해당 함수들을 모두 오버라이드 시켜줘야한다.

```
class Set {
    has(value: number): boolean {
        // ...
    }
    add(value: numver): Set {
        // ...
    }
}

class MutableSet extends Set {
    delete(value: number): boolean {
        // ...
    }
    add(value: number) :MutableSet {
        //...
    }
}
```

위 코드는 아래와 같이 사용하면 간결해진다.

```
class Set {
    has(value: number): boolean {
        // ...
    }
    add(value: numver): this {
        // ...
    }
}

class MutableSet extends Set {
    delete(value: number): boolean {
        // ...
    }
}
// MutableSet.add()는 MutableSet을 반환해준다.
```

4. 인스턴스

타입 별칭과 인터페이스는 문법만 다를뿐 거의 같은 기능을 한다.

```
type Suchi = {
    calories: number,
    salty: boolean,
    tasty: boolean,
}
```

```
interface Sushi = {
    calories: number,
    salty: boolean,
    tasty: boolean,
}
```

인터페이스의 경우 객체 타입, 클래스, 다른 인터페이스 모두를 상속 받을 수 있다.

```
interface Food {
    calories: number,
    tasty: boolean
}

interface Sushi extends Food {
    salty: boolean
}
interface Cake extends Food {
    sweet: boolean
}
```

타입과 인터페이스의 차이

1. 타입 별칭의 경우, 오른편에는 타입 표현식을 포함한 모든 타입이 올 수 있다. 인터페이스의 경우 오른편에는 반드시 형태가 나와야한다.
2. 인터페이스를 상속할 때 타입스크립트는 상속받는 인터페이스의 타입에 상위 인터페이스를 할당할 수 있는지 확인한다.
3. 이름과 범위가 같은 인터페이스가 여러 개 있다면 이들이 자동으로 합쳐진다. ⇒ 선언합침

인터페이스 vs 추상 클래스 상속

- 인터페이스가 더 범용으로 쓰이며 가벼움
- 추상 클래스의 경우 특별한 목적과 풍부한 기능을 갖는다.
- 언제 쓰이는지? 여러 클래스에서 공유하는 구현이라면 추상 클래스를 사용
- 이 클래스는 T다 라고 말하는 것이 목적이라면 인터페이스를 사용하자

5 .클래스의 구조 기반 타입 지원

클래스의 형태를 공유하는 다른 모든 타입은 클래스의 이름이 아닌 구조를 기반으로 비교한다. 그 예시는 아래와 같다.

```
class Zebra {
    trot() {}
}
class Poodle {
    trot() {}
}

function ambleAround(animal: Zebra){
    animal.trot()
}
```

```
let zebra = new Zebra
let poodle = new Poodle

ambleAround(zebra); //Ok
ambleAround(poddle); // Ok
```

단, private이나 protected 필드를 갖는 클래스의 경우 할당하려는 클래스나 서브클래스의 인스턴스가 아니라면 할당 할 수 없다고 판단한다.

10. TS에서의 final

final이란, 클래스나 메서드를 확장하거나 오버라이드 할 수 없게 만드는 기능이다.

- TS에서는 final 키워드를 제공하지 않으나, 비공개 생성자(private constructor)로 final 클래스를 흉내 낼 수 있다.

```
class MessageQueue {
    private constructor(private messages: string[]) {}
}

class BadQueue extends MessageQueue {} //Error

new MessageQueue([]); //Error
```

위 코드의 문제점은 클래스 상속만 막아야하는 상황이지만, 인스턴스화 하는 기능도 막아집니다. 이를 해결하기 위해서는 아래와 같이 class 내부에서 해당 클래스를 인스턴스화하여 반환해주는 함수를 만들어주면 됩니다.

```
class MessageQueue {
    private constructor(private messages: string[]) {}
    static create(messages: string[]) {return new MessageQueue(messages)};
}
```

11. 디자인 패턴

11-1. 팩토리 패턴

어떤 객체를 만들지를 전적으로 팩토리에 위임한다. - 추상화 규칙

호출자는 팩토리가 특정 인터페이스를 만족하는 클래스를 제공할 것이라는 사실만 알 뿐 어떤 구체 클래스가 이 일을 하는지 알 수 없어야 한다.

11-2 빌더 패턴

객체의 생성과 객체 구현 방식을 분리할 수 있는 디자인패턴 방법론