Arrow function과 function의 차이점

this, argument의 바인딩이 다르다.

Arrow Function은 this 바인딩을 갖지 않는다. 기존의 function에서 this의 탐색범위가 함수의 {} 안에서 찾은 반면, Arrow Function에서 this는 일반적인 인자/변수와 동일하게 취급된다.

```
function objFunction(){
    console.log('Inside `objFunction`:', this.foo);
    return{
        foo:25,
        bar: function(){
            console.log('Inside `bar`:', this.foo)
        },
    };
}

objFunction.call({foo:13}).bar();
```

결과는 아래와 같다.

```
Inside `objFunction`: 13 // 처음에 인자로 전달한 값을 받음
Inside 'bar' : 25 // 자신이 있는 object 를 this로 인지해서 25 반환
```

그러나 Arrow Function을 실행하면 이야기가 약간 달라진다.

```
function objFunction(){
    console.log('Inside `objFunction`:', this.foo);
    return {
        foo:25,
        bar: ()=>{
            console.log('Inside `bar`:', this.foo);
        },
    };
}
// objFunction의 `this`를 오버라이딩 합니다.
objFunction.call({foo:13}).bar();
```

위 코드의 결과는 아래와 같다.

```
Inside `objFunction`:13 // 처음에 인자로 전달한 값을 받음
Inside `bar`: 13 // Arrow Function에서 this는 일반 인자로 전달되었기 때문에 이미 값이
```

```
13으로 지정된다.
```

즉, Arrow Function 안의 this는 objFunction의 this가 된다.

Arrow Function은 this의 Scope를 바꾸고 싶지않을 때 유용하다.

```
// ES5 function에서는 `this` Scope가 function안에 들어가면 변하기 때문에 새로운 변수를 만들어 쓰다.
var someVar = this;
getDate(function(data){
    someVar.data = data;
});
// ES6 Arrow Function에서는 `this` Scope의 변화가 없기 때문에 `this`를 그대로 사용하면 된다.
getData(data=>{
    this.data = data;
});
```

정리

함수 정의 방식을 바꿔서 사용할 수 있는 경우는 다음과 같다.

- this나 arguments를 사용하지 않는 경우
- .bind(this)를 사용하는 경우

함수 정의 방식을 바꿔서 사용할 수 없는 경우는 다음과 같다.

- new등을 사용하는 constructable한 함수
- prototype에 덧붙여진 함수나 method들(보통 this를 사용한다.)
- argument를 함수의 인자로 사용한 경우