TypeScript Interface

TypeScript

TypeScript Interface Type

1. Interface를 이용한 Type 설정

TypeScript의 핵심 원리 중 하나는 type-checking이 값의 형태(shape)에 초점을 맞춘다는 것이다. 이것은 때때로 "duck typing" 또는 "structural subtyping" 라고도 한다. TypeScript에서 인터페이스는 이러한 타입의 이름을 지정하는 역할을 하며 외부의 코드와 여러분의 코드의 약속을 정의하는 강력한 방법이다.

Interface의 Type-checker는 프로퍼티가 어떤 순서로 든 상관하지 않으며 인터페이스에 필요한 프로퍼티가 있고 정확한 타입이 있으면 성립된다.

```
interface LabelType {
        label: string;
}
function printLabel(labelObj: LabelType){
        console.log(labelObj.label);
}

let myObj = {size: 10, label: "Size 10 Object"};
printLabel(myObj);
```

TypeScript

TypeScript Interface Type

2. Optional property

인터페이스의 모든 프로퍼티가 필수일 필요는 없다. 일부는 특정 조건 하에서 존재하거나 전혀 존재하지 않을 수도 있다. 이러한 Optional property는 "option bags"와 같이 두 개의 프로퍼티가 있는 객체를 함수에 전달하는 패턴을 생성할 때 많이 사용된다.

Optional property를 가지는 인터페이스는 다른 인터페이스와 비슷하게 작성되며, 단지 프로퍼티 선언의 이름 끝에 ? 가 추가 된다. Optional propery의 장점은 인터페이스의 일부가 아닌 프로퍼티의 사용을 방지하고 사용 가능한 프로퍼티을 열거할 수 있다는 것이다

```
interface LabelType {
    label?: string;
    color?: string;
}
```

TypeScript

TypeScript Interface Type

3. readonly property

일부 프로퍼티는 객체를 처음 만들 때만 수정할 수 있어야 할 수 있습니다. 프로퍼티 이름 앞에 readonly를 붙여 읽기전용으로 지정할 수 있다

TypeScript에는 Array<T>와 동일하지만 모든 수정가능한 메서드가 제거된 ReadonlyArray<T> 타입이 있으므로 배열 생성 후 배열을 변경하지 않도록 할 수 있습니다.

ReadonlyArray<T>은 Type assertion으로 Override 할 수 있다.

```
interface PointType {
        readonly x: number;
        readonly y: number;
}

let pl: PointType = {x: 10, y: 20};                    error => pl.x = 100;
let ro: ReadonlyArray<number> = [10, 11, 100];         error => ro[0] = 20;
let ary: number[] = ro as number[];
```

TypeScript

TypeScript Interface Type assertion

4. property check

객체를 함수의 파라미터로 넘기는 것과 "option bags" 패턴을 결합하여 JavaScript에서와 같은 방법으로 사용할 수 있다.

```
interface SquareConfig {
        color?: string;
        width?: number;
        [propName: string]: any;
}

function createSquare(config: SquareConfig): { color: string; area: number } {
        return { color: "red", area: 10 }
}
```

width 속성은 호환 가능하고 color 속성은 없으며, 기술된 colour 속성은 중요하지 않음.

```
let mySquare = createSquare ( { colour: "red", width: 100 } as SquareConfig );
let mySquare = createSquare ( { colour: "red", width: 100 } );
```

TypeScript

TypeScript Interface Function Type

5. Function Type

인터페이스는 JavaScript 객체가 취할 수 있는 다양한 형태로 설명할 수 있다. 프로퍼티을 가진 객체를 설명하는 것 외에도 인터페이스는 함수 타입을 정의할 수 있다.

TypeScript는 인터페이스에 있는 함수 타입을 정의하기 위해 *Call signature* 인터페이스를 제공한다. 이것은 주어진 파라미터 목록과 리턴 타입만 있는 함수 선언과 같다. 파라미터 목록의 각 파라미터는 이름과 타입이 모두 필요하다.

```
interface AddTeye {
    (x: number, y: number): number;
}

let onAdd: AddTeye = function(kor, eng) {
    return kor + eng;
}
```

TypeScript

TypeScript Interface Indexable Type

6. Indexable Type

함수 타입을 설명하기 위해 인터페이스를 사용하는 방법과 마찬가지로 [10], 또는 ageMap["deniel"] 처럼 "인덱스" 할 수 있는 타입을 정의할 수 있다. 인덱싱 가능 유형에는 인덱싱할 때 대응되는 리턴 타입과 함께 객체에 대해 인덱싱하는 데 사용할 수 있는 타입을 설명하는 Index signature가 있다

지원되는 Index signature에는 문자열과 숫자의 두 가지 타입이 있다. 문자열 Index signature은 “Dictionary” 패턴을 설명하는 강력한 방법이지만 모든 프로퍼티가 리턴 타입과 일치 해야 한다

```
let fruit: { [index: string]: number } = {
        "apple": 10,
        "lemon" : 20
}
interface NumberDictionary {
        [index: string]: number;
        length: number;                 // ok, length is a number
        name: string;                   // error, 'name'의 타입은 인덱서의 하위 타입이 아니다.
}
```

TypeScript

TypeScript Interface Indexable Type

6. Indexable Type

인덱스에 값을 할당하지 못하도록 Index signature을 읽기 전용으로 만들 수 있다.

```
interface ReadonlyStringArray {
        readonly [index: number]: string;
}
let myArray: ReadonlyStringArray = ["Alice", "Bob"];
myArray[2] = "Mallory";                    error!
```

TypeScript

TypeScript Interface Extends

7. Interface Extends

클래스와 마찬가지로 인터페이스는 서로를 확장할 수 있다. 이렇게 하면 한 인터페이스의 구성원을 다른 인터페이스로 복사할 수 있으므로 인터페이스를 재사용 가능한 구성 요소로 분리하는 방법을 보다 유연하게 사용할 수 있다.

```
interface Shape {
    color: string;
}

interface Square extends Shape {
    sideLength: number;
}

let square = <Square>{};
square.color = "blue";
square.sideLength = 10;
```

TypeScript

TypeScript Interface Hybrid Type

8. Hybrid Type

인터페이스는 실제 JavaScript에서 제공되는 풍부한 타입을 나타낼 수 있다. JavaScript의 역동적이고 유연한 특성으로 인해 위에 설명된 일부 타입의 조합으로 작동하는 객체가 종종 필요할 수 있다.

```
interface Counter {
    (start: number): string;
    interval: number,
    reset(): void;
}

function getCounter(): Counter {
    let counter = <Counter>function(start: number){ }
    counter.reset = function() { };
    return counter;
}
```

```
let c = getCounter();
c(10);
c.reset();
c.interval = 1;
```