

# TypeScript\_20210110

## 객체와 타입

타입스크립트는 자바스크립트와 호환성을 위해 자바스크립트의 타입과 타입스크립트의 타입을 모두 지원한다.

### Type

JavaScript	TypeScript
Number	number
Boolean	boolean
String	string
Object	object

### 변수

```
// let: 코드에서 값이 변경도리 수 있음
let num = 123; // num: 123
num = 321; // num: 321
// const: 코드에서 값 변경이 불가능하며 선언시 초기화 필요
const name = 'solchan' // name: solchan
name = 'boy' // name: solchan, 불가능
```

타입스크립트에서 변수를 선언할 때, 타입을 지정할 수 있다.

```
// JavaScript Code
let num: number = 123; // num: 123
num = 321; // num: 321
num = 'one'; // num: 123, string은 불가능

// TypeScript Code
const name: string = 'solchan' // name: solchan
const age: number = '24'; // X, number타입에 string 불가능
```

하지만 타입스크립트는 자바스크립트와 호환성을 위해 아래와 같이 선언하여도 타입 추론이 가능하다. 이를 타입 추론이라고 한다.

```
let num = 123; // 123을 통해 number로 타입 추론
const name = 'solchan' // 'solchan'을 통해 string으로 타입 추론
```

혹은, any타입을 지정할 수 있다. any타입은 타입을 지정하지 않은 경우와 같이 동작한다.

```
let val: any = 123; // 지금은 숫자
val = {}; // 지금은 객체
val = 'solchan'; // 지금은 문자열
```

마지막으로 undefined에 대해 알아보면, 최하위 타입으로 타입스크립트에서는 타입과 값으로 동작한다.

```
let val: undefined = undefined; // 타입: undefined, 값: undefined
val = 1; // 불가능, val 타입이 undefined이기 때문에 number는 불가능
```

## 인터페이스

타입스크립트에서 object 타이비으로 선언된 변수는 number, boolean, string 타입의 값을 가질 수는 없지만, 속성이 다른 object에 대해서는 마치 any 타입처럼 담겨진다.

```
let obj: object = { name: 'solchan'};
obj = { name: 'sochan', age: 24};
```

1번 코드에는 name만을 갖는 objectdlrh, 2번 코드는 name, age를 갖는 object입니다. 서로 속성이 다르지만, object로 타입이 같아 위 코드가 가능하아. 이런 부분을 해결하기 위해 인터페이스 구문이 고안되었고, 아래와 같은 방법으로 사용된다.

```
// 위 인터페이스를 통해 예시를 보면
interface User {
  name: string;
  age: number;
  phone?: string; // 변수명 + ? 는 선택 항목을 뜻한다.
}

let user1: User = { name: 'solchan', age: 24 }
// user1 -> 필수 항목인 name과 age 모두 초기화, 성공!
let user2: User = { name: 'solchan', phone: '123-123' }
// user2 -> 필수 항목인 age가 없음, 오류!
let user3: User = { name: 'solchan', age: 24, location: 'YongIn' }
// user3 -> 필수 항목이 모두 있지만 location은 인터페이스에 없음, 오류!
let user4: User = { name: 'sochan', age: 24, phone: '123-123' }
// user4 -> 필수가 모두 존재하고 phone은 선택으로 존재한다, 성공!
```

익명의 인터페이스는 주로 함수에서 사용된다.

```
let account: { email: string, password: string } = {
  email: 'solchan@interface.hello', password: 'typescript'
}

function printMe( me: { name: string, age: number }) {
  console.log( `name: ${me.name}, age: ${me.age}` )
}

printMe({ name: 'solchan', age: 24});
```

## 클래스

타입스크립트는 C++, JAVA와 같이 객체지향언어이다. 따라서 class, 접근 제한 키워드 private, protected, public 그리고 implements, extend와 같은 키워드를 제공한다.

그리고 constructor라는 생성자가 있는데, 객체 생성을 다음과 같이 할 수 있다.

```
/* 아래 여러 방법으로 다음과 같은 객체를 생성할 수 있다. */
const user: User = new User('solchan', 24);
console.log(user);
// 출력 -> User { name: 'solchan', age: 24}

/* 1번 */
class User {
  constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
  }
  name: string;
  age: number;
}

/* 2번 */
/* 생성자 파라미터 부분에 아래처럼 하여 속성 정의를 생략할 수 있다. */
class User {
  constructor(public name: string, public age: number) {}
}
```

## 인터페이스 구현

```
// 이름과 나이를 정의하는 인터페이스를 상속하는 클래스
interface User {
  name: string;
  age?: number;
}
```

```
class UserClass {
    constructor(public name: string, public age: number){}
}
const user: UserClass = new UserClass('solchan', 24);
console.log(user);
// 출력 -> UserClass { name: 'solchan', age: 24 }
```

interface의 age는 필수가 아닌 선택으로 정의됐지만, 클래스가 인터페이스를 구현하는 경우에는 반드시 속성으로 포함되어야 한다.

타입스크립트는 다른 객체지향 언어처럼 abstract 키워드를 통해 아래와 같이 추상 클래스를 만들어 사용할 수 있다.

```
abstract class AbstractUser {
    abstract name: string;
    constructor(public age: number) {}
}
class user extends AbstractUser{
    constructor(public name: string, age: number){
        super(age); // super()는 부모 클래스의 생성자를 호출한다.
    };
    /* 부모 클래스의 name이 abstract이기 때문에, 구현 구문을 작성한다.
    생성자 구문에 작성하였다. */
}

console.log(new user('solchan', 24));
// 출력 -> user { age: 24, name: 'solchan' }
```

## 구조 분해 할당 (비구조화 할당) / Destructuring

구조 분해 할당은 객체와 배열 모두 사용 가능하다. 여기서 객체에 대한 구조 분해 할당을 알아보자

```
class User {
    constructor(
        public name: string,
        public age: number,
        public location: string){}
}
const user1 = new User('solchan', 24, 'Y');

const { age, name: nickname, location: loc } = user1;
console.log(`이름: ${nickname}, 나이: ${age}, 지역: ${loc}`)
// 출력 -> 이름: solchan 나이: 24 지역: Y
```

User1 객체에서 name을 nickname이라는 이름으로, location을 loc로, age는 그대로 꺼내어 마치 단독 변수처럼 사용할 수 있다.

## 스프레드 연산자

스프레드 연산자는 ...를 통해 필요없는 속성을 빼거나 다른 속성을 추가할 수 있다. 빼는 것을 Rest, 추가하는 것을 spread라고 한다.

```
class User {
  constructor(
    public name: string,
    public age: number,
    address: string,
    country: string){
    this.address = address;
    this.country = country;
  };
  address: string;
  country: string;
}

const user = new User('solchan', 24, 'YongIn', 'Korea');

const { country, address, ...nameAgeUser } = user;
console.log(user);
console.log(nameAgeUser);
/*
출력(user) -> User { name: 'solchan', age: 24, address: 'YongIn', country: 'Korea' }
출력(nameAgeUser) -> { name: 'solchan', age: 24 }
*/
```

16번 Line 코드를 보면 country, address를 user에서 빼고 이를 nameAgeUser로 만드는 동작을 한다. 출력 결과를 보면 nameAgeUser는 country, address가 빠진 상태로 출력됨을 알 수 있다.

user는 User라고 클래스 명을 출력해주지만 nameAgeUser는 country와 address 속성이 빠졌기 때문에 User클래스가 되지 못한다. 따라서 속성만 출력 되는 것을 알 수 있다.

```
const part1: object = {
  name: 'solchan'
}
```

```
const part2: object = {
  country: 'Korea',
  address: 'YongIn',
}

const user = { ...part1, ...part2 };
console.log(part1);
console.log(part2);
console.log(user);
// 출력(part1) -> { name: 'solchan' }
// 출력(part2) -> { country: 'Korea', address: 'YongIn' }
// 출력(user) -> { name: 'solchan', country: 'Korea', address: 'YongIn' }
```

10번 Line 코드를 보면 part1, part2를 합하여 user를 만드는 것을 알 수 있다.