



정적 타이핑

타입 선언 (Type Declaration)

TypeScript는 아래와 같이 변수명 뒤에 타입을 명시하는 것으로 타입을 선언할 수 있다.

```
// 변수 foo는 string 타입이다.  
let foo: string = 'hello';
```

선언한 타입에 맞지 않는 값을 할당하면 컴파일 시점에 에러가 발생한다.

```
let bar: number = true; // error TS2322: Type 'true' is not  
assignable to type 'number'.
```

이러한 타입 선언은 개발자가 코드를 예측할 수 있도록 돕는다. 또한 타입 선언은 강력한 타입 체크를 가능하게 하여 문법 에러나 타입과 일치하지 않는 값의 할당 등 기본적인 오류를 런타임 이전에 검출한다. 비주얼스튜디오 코드(VS Code)와 같은 도구를 사용하면 코드를 작성하는 시점에 에러를 검출할 수 있어서 개발 효율이 대폭 향상된다.

```
TS app.ts  
  
1 let foo: string = 'hello';  
2 let bar: number = true;
```

함수의 매개변수와 반환값에 대한 타입 선언 방법은 아래와 같다. 일반 변수와 마찬가지로 선언된 타입에 일치하지 않는 값이 주어진다면 에러가 발생한다.

```
// 함수선언식
function multiply1(x: number, y: number): number {
    return x * y;
}

// 함수표현식
const multiply2 = (x: number, y: number): number => x * y;

console.log(multiply1(10, 2));
console.log(multiply2(10, 3));

console.log(multiply1(true, 1)); // error TS2345: Argument
of type 'true' is not assignable to parameter of type 'number'.
```

TypeScript는 ES5, ES6의 Superset(상위확장)이므로 자바스크립트의 타입을 그대로 사용할 수 있다. 자바스크립트의 타입 이외에도 TypeScript 고유의 타입이 추가로 제공된다.

Type	JS	TS	Description
boolean	○	○	true와 false
null	○	○	값이 없다는 것을 명시
undefined	○	○	값을 할당하지 않은 변수의 초기값
number	○	○	숫자(정수와 실수, Infinity, NaN)
string	○	○	문자열
symbol	○	○	고유하고 수정 불가능한 데이터 타입이며 주로 객체 프로퍼티들의 식별자로 사용(ES6에서 추가)
object	○	○	객체형(참조형)
array		○	배열
tuple		○	고정된 요소수 만큼의 타입을 미리 선언후 배열을 표현

Type	JS	TS	Description
enum		○	열거형. 숫자값 집합에 이름을 지정한 것이다.
any		○	타입 추론(type inference)할 수 없거나 타입 체크가 필요없는 변수에 사용. var 키워드로 선언한 변수와 같이 어떤 타입의 값이라도 할당 가능.
void		○	일반적으로 함수에서 반환값이 없을 경우 사용한다.
never		○	결코 발생하지 않는 값

다양한 타입을 사전 선언하는 방법은 아래와 같다.

```
// boolean
let isDone: boolean = false;

// null
let n: null = null;

// undefined
let u: undefined = undefined;

// number
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;

// string
let color: string = "blue";
color = 'red';
let myName: string = `Lee`; // ES6 템플릿 문자열
let greeting: string = `Hello, my name is ${ myName }.`; // ES6 템플릿 대입문
```

```

// object
const obj: object = {};

// array
let list1: any[] = [1, 'two', true];
let list2: number[] = [1, 2, 3];
let list3: Array<number> = [1, 2, 3]; // 제네릭 배열 타입

// tuple : 고정된 요소수 만큼의 타입을 미리 선언후 배열을 표현
let tuple: [string, number];
tuple = ['hello', 10]; // OK
tuple = [10, 'hello']; // Error
tuple = ['hello', 10, 'world', 100]; // Error
tuple.push(true); // Error

// enum : 열거형은 숫자값 집합에 이름을 지정한 것이다.
enum Color1 {Red, Green, Blue};
let c1: Color1 = Color1.Green;

console.log(c1); // 1

enum Color2 {Red = 1, Green, Blue};
let c2: Color2 = Color2.Green;

console.log(c2); // 2

enum Color3 {Red = 1, Green = 2, Blue = 4};
let c3: Color3 = Color3.Blue;

console.log(c3); // 4

/*
any: 타입 추론(type inference)할 수 없거나 타입 체크가 필요 없는 변수에 사용한다.
var 키워드로 선언한 변수와 같이 어떤 타입의 값이라도 할당할 수 있다.
*/
let notSure: any = 4;

```

```

notSure = 'maybe a string instead';
notSure = false; // okay, definitely a boolean

// void : 일반적으로 함수에서 반환값이 없을 경우 사용한다.
function warnUser(): void {
    console.log("This is my warning message");
}

// never : 결코 발생하지 않는 값
function infiniteLoop(): never {
    while (true) {}
}

function error(message: string): never {
    throw new Error(message);
}

```

타입은 소문자, 대문자를 구별하므로 주의가 필요하다. 위에서 살펴본 바와 같이 TypeScript가 기본 제공하는 타입은 모두 소문자이다. 아래 코드를 살펴보자.

```

// string: 원시 타입 문자열 타입
let primitiveStr: string;
primitiveStr = 'hello'; // OK
// 원시 타입 문자열 타입에 객체를 할당하였다.
primitiveStr = new String('hello'); // Error
/*
Type 'String' is not assignable to type 'string'.
'string' is a primitive, but 'String' is a wrapper object.
Prefer using 'string' when possible.
*/

// String: String 생성자 함수로 생성된 String 래퍼 객체 타입
let objectStr: String;
objectStr = 'hello'; // OK
objectStr = new String('hello'); // OK

```

string 타입은 TypeScript가 기본으로 제공하는 원시 타입인 문자열 타입을 의미한다. 하지만 대문자로 시작하는 String 타입은 String 생성자 함수로 생성된 String 래퍼 객체 타입을 의미한다. 따라서 string 타입에 String 타입을 할당하면 에러가 발생한다. 하지만 String 타입에는 string 타입을 할당할 수 있다. 이처럼 객체의 유형도 타입이 될 수 있다.

```
// Date 타입
const today: Date = new Date();

// HTMLElement 타입
const elem: HTMLElement = document.getElementById('myId');

class Person { }
// Person 타입
const person: Person = new Person();
```

정적 타이핑

C나 Java같은 C-family 언어는 변수를 선언할 때 변수에 할당할 값의 타입에 따라 사전에 타입을 명시적으로 선언(Type declaration)하여야 하며 선언한 타입에 맞는 값을 할당해야 한다. 이를 정적 타이핑(Static Typing)이라 한다.

자바스크립트는 동적 타입(dynamic typed) 언어 혹은 느슨한 타입(loosely typed) 언어이다. 이것은 변수의 타입 선언 없이 값이 할당되는 과정에서 동적으로 타입을 추론(Type Inference)한다는 의미이다. 동적 타입 언어는 타입 추론에 의해 변수의 타입이 결정된 후에도 같은 변수에 여러 타입의 값을 교차하여 할당할 수 있다. 이를 동적 타이핑(Dynamic Typing)이라 한다.

동적 타이핑은 사용하기 간편하지만 코드를 예측하기 힘들어 예상치 못한 오류를 만들 가능성이 높다. 또한 IDE와 같은 도구가 변수나 매개 변수, 함수의 반환값의 타입을 알 수 없어 코드 어시스트 등의 기능을 지원할 수 없게 한다.

```
var foo;

console.log(typeof foo); // undefined
```

```

foo = null;
console.log(typeof foo); // object

foo = {};
console.log(typeof foo); // object

foo = 3;
console.log(typeof foo); // number

foo = 3.14;
console.log(typeof foo); // number

foo = "Hi there";
console.log(typeof foo); // string

foo = true;
console.log(typeof foo); // boolean

```

TypeScript의 가장 독특한 특징은 **정적 타이핑**을 지원한다는 것이다. 정적 타입 언어는 타입을 명시적으로 선언하며, 타입이 결정된 후에는 타입을 변경할 수 없다. 잘못된 타입의 값이 할당 또는 반환되면 컴파일러는 이를 감지해 에러를 발생시킨다.

```

let foo: string,    // 문자열 타입
    bar: number,    // 숫자 타입
    baz: boolean;   // 불리언 타입

foo = 'Hello';
bar = 123;
baz = 'true'; // error: Type '"true"' is not assignable to type 'boolean'.

```

정적 타이핑은 변수는 물론 함수의 매개변수와 반환값에도 사용할 수 있다.

```

function add(x: number, y: number): number {
  return x + y;
}

```

```
console.log(add(10, 10)); // 20
console.log(add('10', '10'));
// error TS2345: Argument of type '"10"' is not assignable
to parameter of type 'number'.
```

참고로 정적 타이핑과 동적 타이핑 중 무엇이 우위인지에 대한 논쟁은 사실 큰 의미가 없다. 정적 타이핑과 동적 타이핑의 가장 큰 차이를 컴파일 시의 에러 검출과 런타임 시의 에러 검출로 볼 수 있는데 Java와 같은 정적 타이핑 언어도 런타임에만 검출되는 에러가 존재하기 때문이다.

정적 타이핑의 장점은 **코드 가독성, 예측성, 안정성의 향상**이라고 볼 수 있는데 이는 대규모 프로젝트에 매우 적합하다.

타입 추론

만약 타입 선언을 생략하면 값이 할당되는 과정에서 동적으로 타입이 결정된다. 이를 타입 추론(Type Inference)이라 한다.

```
let foo = 123; // foo는 number 타입
```

위 코드를 보면 변수 foo에 타입을 선언하지 않았으나 타입 추론에 의해 변수의 타입이 결정된다. 동적 타입 언어는 타입 추론에 의해 변수의 타입이 결정된 후에도 같은 변수에 여러 타입의 값을 교차하여 할당할 수 있다. 하지만 정적 타입 언어는 타입이 결정된 후에는 타입을 변경할 수 없다.

TypeScript는 정적 타입 언어이므로 타입 추론으로 타입이 결정된 이후, 다른 타입의 값을 할당하면 에러가 발생한다.

```
let foo = 123; // foo는 number 타입
foo = 'hi';    // error: Type '"hi"' is not assignable to t
ype 'number'.
```

타입 선언을 생략하고 값도 할당하지 않아서 타입을 추론할 수 없으면 **any** 타입이 된다. **any** 타입의 변수는 자바스크립트의 var 키워드로 선언된 변수처럼 어떤 타입의 값도 재


할당이 가능하다. 이는 TypeScript를 사용하는 장점을 없애기 때문에 사용하지 않는 편이 좋다.

```
let foo; // let foo: any와 동치

foo = 'Hello';
console.log(typeof foo); // string

foo = true;
console.log(typeof foo); // boolean
```

타입 캐스팅

기존의 타입에서 다른 타입으로 타입 캐스팅하려면 `as` 키워드를 사용하거나  연산자를 사용할 수 있다. 다음 예제를 살펴보자.

```
const $input = document.querySelector('input["type="text"]');
// => $input: Element | null

const val = $input.value;
// TS2339: Property 'value' does not exist on type 'Element'.
```

`document.querySelector` 메서드는 `Element | null` 타입의 값을 반환한다. 따라서 위 예제의 `$input`은 `Element | null` 타입이다. 이때 `$input.value`를 실행하면 `Element` 또는 `null` 타입에는 `value`이란 프로퍼티가 존재하지 않으므로 컴파일 에러가 발생한다.

`value` 프로퍼티는 `Element` 타입의 하위 타입인 `HTMLInputElement` 타입에만 존재한다. 따라서 다음과 같이 타입 캐스팅이 필요하다.

```
const $input = document.querySelector('input["type="text"]') as HTMLInputElement;
const val = $input.value;
```

또는 타입 캐스팅을 위해 `as` 키워드 대신 `<>` 연산자를 사용할 수도 있다.

```
const $input = <HTMLInputElement>document.querySelector('input["type="text"]');  
const val = $input.value;
```