타입스크립트

WHATEVER YOU WANT, MAKE IT REAL.

강사 정길용

LIKE LION

타입스크립트란?

개발 환경 구축

{ 기본 타입 }

타입 별칭

인터페이스

제네릭

기타 문법

타입스크립트란?

- 자바스크립트에 타입을 부여한 언어
- 공식 URL: https://www.typescriptlang.org
- 마이크로소프트에서 개발
- 자바스크립트 기반의 강형(strongly typed) 프로그래밍 언어
- 타입스크립트로 작성된 코드는 TypeScript Compiler(tsc)를 이용해 자바스크립트로 컴파일 되며 이때 타입 체크가 이루어지고 잘못된 타입을 사용한 경우 컴파일 에러가 발생

```ts
function sum(a: number, b: number): number {
  return a + b;
}
sum(10, 20);
```

TS

```js
function sum(a, b) {
  return a + b;
}
sum(10, 20);
```

JS

▶ 타입스크립트의 장점

- 에러 사전 검출
- 코드 가이드 및 자동 완성

JS

```
function sum(a, b) {
  return a + b;
}

sum(10, 20);    // 30
sum(10, '20'); // 1020
```

TS

```
function sum(a: number, b: number): number{
  return a + b;
}

sum(10,
sum(10, '20');
```

Argument of type 'string' is not assignable to parameter of type 'number'. ts(2345)

View Problem (Alt+F8) No quick fixes available

TS

```
function sum(a: number, b: number): number{
  return a + b;
}

sum(10, 20).;
sum(10, '20'
```

- toExponential
- toFixed
- toLocaleString
- toPrecision
- toString
- valueOf

(method) Number.toFixed(fractionDigits?: number | u
ndefined): string

Returns a string representing a number in fixed-point notation.

@param `fractionDigits` — Number of digits after the decimal point. Must be in the range 0 - 20, inclusive.

TypeScript Compiler(tsc)

- 타입 검사
  - 타입 검사에서 오류를 찾으면 컴파일 에러가 발생하므로 런타임이 아닌 컴파일 타임에 에러 검출
- 컴파일(트랜스파일)
  - 타입스크립트 소스 코드를 자바스크립트 소스 코드로 변환
  - 타입 관련 구문 제거
  - 브라우저 호환성을 위해 지정한 구(old) 버전의 자바스크립트로 변환

TS

```ts
class Score {
  kor: number;
  eng: number;
  constructor(kor: number, eng: number) {
    this.kor = kor;
    this.eng = eng;
  };
  sum(){
    return this.kor + this.eng;
  };
}
```

JS

```js
var Score = /** @class */ (function () {
  function Score(kor, eng) {
    this.kor = kor;
    this.eng = eng;
  };
  Score.prototype.sum = function () {
    return this.kor + this.eng;
  }
  return Score;
}());
```

tsc --target es5 명령으로 컴파일

LIKE LION

ex06-01.ts

- Node.js
  - https://nodejs.org/en/download
- VSCode
  - https://code.visualstudio.com/download
- 최신 버전의 웹브라우저
  - Chrome
  - Safari
- TypeScript Compiler
  - npm i typescript -g
- 컴파일
  - tsc ex06-01.ts
- 실행
  - node ex06-01.js

```ts
function hello(name: string): string {
  return 'Hello ' + name;
}
console.log(hello('TypeScript'));
```

```js
function hello(name) {
  return 'Hello ' + name;
}
console.log(hello('TypeScript'));
```

LIKE LION

ex06-02.ts 

기본 타입

- string
- number
- boolean
- null
- undefined
- bigint
- symbol

참조 타입

- object
- Array<T>
- tuple

```ts
let str: string = 'hello';
str = 123; // 컴파일 에러
let age: number = 30;
let done: boolean = true;
let nullVal: null = null;
let emptyVal: undefined;
let todo: object = { title: 'TypeScript 공부', done: false };
let todoList: Array<string> = ['JavaScript 공부', 'TypeScript 공부'];
let todoList2: string[] = ['React 공부', 'Next.js 프로젝트'];
let items: [string, number] = ['타스핑', 9];
let userName: any = '이일구';
userName = 219;
```

특수 타입

- any: 모든 타입 허용(비추천)
- unknown: 모든 타입을 허용하지만 사용 전 타입 검사가 필수이므로 any 보다 안전

LIKE LION

ex06-03.ts 

매개 변수의 타입 지정

- 매개 변수명 뒤에 지정

리턴 타입 지정

- 매개 변수 선언부 뒤에 지정
- 리턴 값이 없을 경우에는 void 지정

```
function getCount(count: number): string {
  return 'Count: ' + count;
}
console.log(getCount(20));
getCount('30'); // 컴파일 에러(count 인자값이 number가 아님)
const count: number = getCount(40); // 컴파일 에러(count 변수가 string이 아님)
getCount(); // 컴파일 에러(count 인자값이 없음)
getCount(10, 20); // 컴파일 에러(인자값이 하나만 있어야 함)
```

LIKE LION

ex06-04.ts

optional parameter

- 함수의 매개 변수를 선택적으로 전달 받고 싶을때 매개 변수명 뒤에 ?를 추가

```ts
function user(name: string, age?: number): void {
  console.log(name, age);
}

user('조아핑', 9);
user('방글핑');
```

```
조아핑 9
방글핑 undefined
```

LIKE LION

ex06-05-01.ts, 02

- union type
  - 여러 종류의 타입을 허용하기 위해 | (OR 연산자) 로 연결한 타입
  - any 타입은 모든 타입을 허용하지만 유니언 타입은 | 연산자로 연결된 타입중 하나를 허용

```
function log(msg: number | string): void {
  console.log(msg);
}

log('hello');
log(200);
```

LIKE LION

ex06-06.ts

type alias

- 값을 변수에 저장하듯, 타입을 변수에 저장해서 사용
- 유니언 타입 같은 복잡한 타입에 의미 있는 이름을 붙여서 사용(별칭)
- type 키워드로 선언하는 사용자 정의 타입
- 동일한 이름으로 중복 선언 불가
- JS로 컴파일 되면 제거됨

```ts
function log(msg: number | string): void {
  console.log(msg);
}
const msg3: string | number = 'world';
const msg4: string | number = 200;
log(msg3, msg4);
```

TS

```ts
type Message = string | number;
function log(msg: Message): void {
  console.log(msg);
}
const msg3: Message = 'world';
const msg4: Message = 200;
log(msg3, msg4);
```

JS

```js
function log(msg) {
  console.log(msg);
}
const msg3 = 'world';
const msg4 = 200;
log(msg3, msg4);
```

LIKE LION

ex06-07.ts 

▶ 타입 별칭으로 객체의 타입 선언

- 객체의 속성명과 속성값의 타입을 지정
- 속성은 , 또는 ; 으로 구분
- 타입 별칭을 타입으로 지정한 객체는 타입 별칭에 정의된 속성명과 속성의 타입을 준수해야 함

```
type User = {
  name: string,
  age: number;
}

// 객체 생성
const ping1: User = {name: '유저핑', age: 30};
const ping2: User = {name: '유저핑'}; // 컴파일 에러(age 속성이 없음)
const ping3: User = {name: '유저핑', age: '30'}; // 컴파일 에러(age 속성값이 number가 아님)
const ping4: User = {name: '유저핑', userAge: 30}; // 컴파일 에러(age 속성이 없음)
```

LIKE LION

ex06-08.ts

- intersection type
  - 타입 여러개를 하나로 합치기 위해 **&** (AND 연산자) 로 연결한 타입
    - 타입 별칭을 확장할 때 주로 사용

```
type Todo = {
  title: string;
  content: string;
}

// Todo 타입 확장
type TodoInfo = Todo & {
  id: number;
  done: boolean;
};
```

LIKE LION

ex06-09.ts 

▶ 인터페이스

- 객체의 타입을 정의하기 위해 사용(객체의 속성명과 속성값의 타입을 지정)
- 속성은 , 또는 ; 으로 구분
- interface 키워드로 선언하는 사용자 정의 타입
- 인터페이스를 타입으로 지정한 객체는 해당 인터페이스에 정의된 속성명과 속성의 타입을 준수해야 함
- JS로 컴파일 하면 제거됨

```
interface User {
  name: string,
  age: number;
}

// 객체 생성
const ping1: User = {name: '유저핑', age: 30};
const ping2: User = {name: '유저핑'}; // 컴파일 에러(age 속성이 없음)
const ping3: User = {name: '유저핑', age: '30'}; // 컴파일 에러(age 속성값이 number가 아님)
const ping4: User = {name: '유저핑', userAge: 30}; // 컴파일 에러(age 속성이 없음)
```

LIKE LION

ex06-10.ts

▶ 인터페이스 사용

- 변수, 함수의 매개 변수, 함수의 리턴 타입에 사용

```ts
interface User {
  name: string;
  age: number;
}
// 변수
const ping: User = { name: '유저핑', age: 30 };
// 함수의 매개 변수
const getAge = (ping: User) => {
  return ping.age;
}
// 함수의 리턴 타입
const createUser = (name: string, age: number): User => {
  return { name, age };
}
```

LIKE LION

ex06-11.ts

인터페이스 사용

- 클래스의 타입 지정에 사용
- 클래스명 뒤에 implements 키워드 추가
- 인터페이스를 타입으로 지정한 클래스의 멤버 변수와 메서드는 인터페이스에 정의된 속성과 속성의 타입을 준수해야 함

```
interface Score {
  kor: number;
  eng: number;
  sum(): number;
  avg(): number;
}
class HighSchool implements Score {
  ...
}
const rara = new HighSchool(100, 90);
console.log(rara.sum(), rara.avg())
```

▶ 인터페이스에 정의하는 속성의 종류

- 객체의 속성과 속성의 타입, 메서드의 매개 변수와 리턴 타입 정의
- 표현식 함수의 매개 변수와 리턴 타입 정의(call signature)
  - 해당 인터페이스가 지정된 함수 생성시 매개 변수와 리턴 타입을 명시할 필요 없음

```
interface User {
  name: string;
  age: number;
  add(year: number): number;
}
interface SaySome {
  (name: string, message: string): string;
}
const say: SaySome = (name, message) => {
  return `${name}이 ${message}라고 말했다.`;
}
```

optional property

- 객체의 속성을 선택적으로 부여하고 싶을때 인터페이스 속성명 뒤에 ?를 추가

```
interface Todo {
  title: string;
  content: string;
  done?: boolean;
}
const todo1: Todo = {
  title: '할일1',
  content: '인터페이스 사용',
  done: true
};
const todo2: Todo = {
  title: '할일2',
  content: 'done 생략'
};
```

LIKE LION

ex06-12.ts

readonly

- 인터페이스의 속성명앞에 readonly 키워드 추가
- 객체 생성시에만 값 할당이 가능하고 생성된 이후에는 수정할 수 없는 속성을 만들때 사용

```
interface Todo {
  readonly id: number;
  title: string;
  content: string;
  done?: boolean;
}
const todo1: Todo = { id:1, title:'할일1', content:'인터페이스 사용', done:true };
todo1.content = '수정함';
todo1.id = 2; // 컴파일 에러(id는 readonly 이므로 수정 불가)

const todo2: Todo = { id: 2, title: '할일2', content: 'done 생략' };
```

LIKE LION

ex06-14.ts

인터페이스 상속

- TS 문법
- 부모 인터페이스의 속성과 메서드 정의를 자식 인터페이스가 물려 받고 확장
- interface 선언부의 extends 키워드 뒤에 상속 받을 부모 인터페이스 지정

```
interface Todo {
  title: string;
  content: string;
}

interface TodoInfoType extends Todo {
  id: number,
  done: boolean
}
```

LIKE LION

ex06-15.ts

▶ 계층 구조로 상속

- 인터페이스 상속은 여러 단계의 계층 구조로 구성 가능

```ts
interface Todo {
  title: string;
  content: string;
}
interface TodoInfo extends Todo {
  id: number;
  done: boolean;
}
interface TodoInfoWithTime extends TodoInfo {
  createdAt: Date;
  updatedAt: Date;
}
```

LIKE LION

ex06-16.ts

▶ 다중 상속

- 둘 이상의 인터페이스를 상속 받음

```
interface Todo {
  title: string;
  content: string;
}
interface TodoList {
  id: number;
  title: string;
  done: boolean;
}
interface TodoInfo extends Todo, TodoList {
  ......
}
```

LIKE LION

ex06-17.ts

인터페이스 재선언(선언 병합)

- 동일한 이름의 인터페이스를 중복으로 선언
- 기존 인터페이스에 없는 속성을 추가해서 확장

```ts
interface Todo {
  title: string;
  content: string;
}

// Todo 인터페이스 선언 병합
interface Todo {
  id: number;
  done: boolean;
};
```

```ts
type Todo = {
  title: string;
  content: string;
}

// 인터섹션 타입을 이용한 Todo 타입 별칭 확장
type TodoInfo = Todo & {
  id: number;
  done: boolean;
};
```

정의할 수 있는 타입 종류

- 타입 별칭: 객체, 클래스, 기본 타입, 유니언 타입, 인터섹션 타입, 유틸리티 타입, 맵드 타입 등의 정의에 사용
- 인터페이스: 객체, 클래스의 타입 정의

타입 확장

- 타입 별칭: & 연산자로 확장(인터섹션 타입)
- 인터페이스: extends 키워드로 확장, 선언 병합

타입 별칭이 필요한 경우

- 객체의 타입을 지정하는 경우 확장이 용이한 인터페이스 사용을 권장
- 객체가 아닌 타입 별칭으로만 정의할 수 있는 경우에만 타입 별칭 사용을 권장

LIKE LION

ex06-18-01.ts, 02 

▶ 제네릭이란?

- 함수를 생성할 때 함수에서 사용할 매개 변수, 리턴 타입을 정의하지 않고 호출하는 시점에 원하는 타입을 지정해서 사용
- 함수 내부의 코드는 동일하고 매개 변수나 리턴 타입만 다를 경우 제네릭 문법을 이용하면 하나의 함수에서 구현 가능

```
function echoString(msg: string): string{
  return msg;
}
function echoNumber(msg: number): number{
  return msg;
}
function echoBoolean(msg: boolean): boolean{
  return msg;
}
console.log(echoString('hello'));
console.log(echoNumber(100));
console.log(echoBoolean(true));
```

```
function echo<T>(msg: T): T {
  return msg;
}

console.log(echo<string>('world'));
console.log(echo<number>(200));
console.log(echo<boolean>(false));
console.log(echo<string>(300)); // 에러
```

LIKE LION

ex06-19-01.js, 02.ts, 03, 04

▶ 제네릭 사용처

- 함수, 인터페이스, 클래스 정의에 사용

```ts
function echo<T>(msg: T): T {
  return msg;
}
```

```ts
interface DropdownItem<T> {
  value: T;
  selected?: boolean;
}
```

```ts
class Queue<T> {
  data: T[] = [];
  push(item: T) {
    this.data.push(item);
  }
  pop(): T | undefined {
    return this.data.shift();
  }
}
```

LIKE LION

ex06-18-03

▶ 제네릭 타입 제약

- 제네릭에 전달받을 타입을 지정한 타입만 가능하도록 제약

▶ extends 키워드 사용

- `<T extends string | number>`
- `<T extends { length: number }>`

```
function echo<T extends string | number>(msg: T): T {
  return msg;
}
function echo2<T extends { length: number }>(msg: T): T {
  console.log('msg.length: ', msg.length);
  return msg;
}
console.log(echo<string>('hello'));
console.log(echo<number>(100));
console.log(echo2<string>('world'));
console.log(echo2<number[]>([200, 300]));
```

LIKE LION

ex06-21.ts 

타입 추론이란?

- 명시적으로 타입을 지정하지 않아도 타입스크립트가 코드를 해석해서 적절한 타입을 자동으로 정의

변수의 타입 추론

- 할당된 값과 일치하는 타입
- 선언만 된 상태라면 any 타입
  - any 타입으로 추론된 이후에 값을 할당해도 타입이 변경되지 않음
  - 선언 시점을 기반으로 추론

```
const name = '이일구'; // string
const name = 219; // error
const age = 20; // number
let address; // any
address = '서울시';
address = 20;
```

LIKE LION

ex06-22.ts

매개 변수의 타입

- 매개 변수에 타입을 지정하지 않으면 **any** 타입
- 기본값 매개 변수를 지정했을 경우 할당된 값과 일치하는 타입으로 추론하고 ?(옵셔널 매개변수) 추가

리턴 타입

- 리턴값을 기반으로 추론
- 매개 변수가 리턴값에 영향을 미치면 매개 변수의 타입과 연산자를 기반으로 리턴값의 타입을 추론
- 아래의 예시에서 num + 20은 number + number 이므로 결과도 number로 추론

```
(local function) add20(num?: number): number
function add20(num = 10){
  return num + 20;
}
```

LIKE LION

ex06-23.ts, 24-01.js, 24-02.ts

type assertion

- 타입스크립트의 타입 추론에 기대지 않고 명시적으로 직접 타입을 지정
- **as** 키워드로 타입을 지정하면 타입스크립트 컴파일러가 타입 검사를 수행하지 않음

타입 단언 대상

- 리터럴: 원시형 데이터 타입의 값, 객체, 함수의 리턴값

타입 단언 중첩

- 타입 단언 중첩 가능

```
const a = 10 as any as string;
```

타입 단언 주의 사항

- as 키워드는 변수에는 지정할 수 없고 데이터(값)에만 지정 가능
- 호환되는 타입으로만 타입 단언이 가능

```
const a = 10 as string; // error
```

- any로 타입 단언을 하면 TSC가 타입 검사를 하지 않기 때문에 런타임 오류 발생 가능성이 높아지므로 남용 금지(타입 안정성이 높은 unknown 권장)
- 가능하다면 타입은 선언해서 사용하고 꼭 필요한 경우에만 타입 단언 사용을 권장

LIKE LION

ex06-25.ts, 26, 27, 28, 29

▶ 타입 가드란?

- 함수의 매개 변수로 여러 종류의 타입이 지정되었을 경우(유니온 타입) 정확한 타입 추론을 할수 있도록 TSC에 힌트를 주는 구문
- 주로 조건문을 이용하고 TSC가 조건문의 구문을 인식해서 조건문 내부에서 만큼은 적절한 타입으로 추론할 수 있도록 도와주는 문법

▶ 타입 가드 구문

- null, undefined 확인
- 논리연산자(&&)
- typeof 연산자
- instanceof 연산자
- in 연산자
- 구별된 유니언 타입(discriminated unions)
  - 타입의 속성 정의시 구체적인 값을 지정한 후 객체의 속성값으로 확인
  - admin: boolean 대신 admin: true
- 타입 가드 함수 작성
  - is 연산자를 사용해서 타입 가드 기능을 하도록 만든 함수

LIKE LION

ex06-30.ts 

타입 호환이란?

- 두 타입이 서로 대입 가능한지 여부를 판단하는 규칙
- 타입 호환여부는 구조적 타입 시스템(Structural Type System)에 기반
  - 구조적인 관점에서 호환 여부를 판단
  - 두 타입이 호환되려면 한 타입이 다른 타입의 모든 프로퍼티와 메서드를 포함해야 함
- 호환 가능 대상
  - 인터페이스: 프로퍼티 비교
  - 함수: 인자의 수와 타입, 리턴 타입을 기준으로 판단
  - 클래스, 제네릭

```
interface Member { id: number; name: string; age: number; }
interface Guest { name: string; }

const rara: Member = {
  id: 1,
  name: '라라핑',
  age: 6
};

const guest: Guest = rara;
```