**ECMA6**

// ES6 -> ES5 트랜스 컴파일러

// - Console 웹사이트 : https://es6console.com

// - Traceur : http://bit.ly/1IE4UJs

// - Bable : http://babeljs.io/repl

//---------------------------------------

// 템플릿 리터럴

//---------------------------------------

function world(){ return 'world'; }

console.log(`Hello ${world()}`);

// 태그가 붙은 템플릿 스트링

var nick = 'vulcan';

let msg = myTag`Hello ${world()}, ${nick}!`;

// Hello - world, vulcan - !

console.log(msg);

function myTag(str, world, name){

return `${str[0]} - ${world}${str[1]} ${name} - ${str[2]}`

}

//---------------------------------------

// 인자 기본값 (함수도 사용 가능)

//---------------------------------------

function method (a, b='b'){ return 0; }

function method (a, b=defaultValue()){ return 0; }

function defaultValue(){ return 'default'; }

// defaultValue 함수는 method1 함수가 실행될 때마다 실행되므로 성능에 신경 써야함

// 옵션 인자

function method1 (a:any, b:string='b', c?:string):number{ return 0; }

// 변수 스코프 (let, const)

// - const를 최대한 사용하자

//---------------------------------------

// 람다 표현식

//---------------------------------------

let sum = (arg1, arg2) => arg1 + arg2;

(arg1, arg2) => {

return arg1 + arg2;

}

() => '전달인자가 없으면 빈 괄호를 사용한다.';

arg1 => '전달인자가 1개이면 괄호를 생략할 수 있다.';

Arrow function에는 함수 객체의 arguments 프로퍼티가 없다.

따라서 화살표 함수에서 가변 인자 함수를 구현해야하는 경우, 반드시 rest 파라미터를 사용하여야 한다.

//---------------------------------------

// 연산자

//---------------------------------------

// 나머지 연산자 (rest operator)

function restOpt(...args){

// args는 배열처럼 사용 : args[0]

}

// 전개 연산자 (spread operator)

let args = [1,2,3];

spreadOpt(...args, '배열을 인자로 사용');

function spreadOpt(arg1, arg2, arg3, stringArg) {

// args 배열이 매개변수로 순서대로 전달됨

}

console.log(...[1, 2, 3]) // -> 1, 2, 3

console.log(...'Helllo'); // H e l l l o

console.log(...new Map([['a', '1'], ['b', '2']])); // [ 'a', '1' ] [ 'b', '2' ]

console.log(...new Set([1, 2, 3])); // 1 2 3

const o1 = { x: 1, y: 2 };

const o2 = { ...o1, z: 3 }; // { x: 1, y: 2, z: 3 }

// 나머지 연산자는 함수 선언 마지막에만 사용

// 전개 연산자는 위치 상관없이 사용

//---------------------------------------

// 이터러블

//---------------------------------------

// Symbol.iterator를 프로퍼티 key로 사용한 메소드를 구현하여야 한다.

// 배열에는 Array.prototype[Symbol.iterator] 메소드가 구현되어 있다.

const iterable = ['a', 'b', 'c'];

// 이터레이터

// 이터러블의 Symbol.iterator를 프로퍼티 key로 사용한 메소드는 이터레이터를 반환한다.

const iterator = iterable[Symbol.iterator]();

// 이터레이터는 순회 가능한 자료 구조인 이터러블의 요소를 탐색하기 위한 포인터로서 value, done 프로퍼티를 갖는 객체를 반환하는 next() 함수를 메소드로 갖는 객체이다. 이터레이터의 next() 메소드를 통해 이터러블 객체를 순회할 수 있다.

console.log(iterator.next()); // { value: 'a', done: false }

console.log(iterator.next()); // { value: 'b', done: false }

console.log(iterator.next()); // { value: 'c', done: false }

console.log(iterator.next()); // { value: undefined, done: true }

ES6에서 제공하는 빌트인 이터러블

Array : Array.prototype[Symbol.iterator]

String : String.prototype[Symbol.iterator]

Map : Map.prototype[Symbol.iterator]

Set : Set.prototype[Symbol.iterator]

DOM data structures : NodeList.prototype[Symbol.iterator], HTMLCollection.prototype[Symbol.iterator]

//---------------------------------------

// 제네레이터

//---------------------------------------

// 함수 실행중 몇번이고 멈출 수있다.

// 멈춘 시점에서 다시 이어서 실행할 수도 있다.

// 다른 제네레이터 함수에게 조작 권한을 넘겨줄 수도 있다.

// 에스테리크(\*)를 붙여 선언

function\* method(){

console.log('실행 시작 후 멈춤');

// yield 키워드를 만나면 함수 실행이 멈춤

// 제네레이터 외부에서 next()를 호출하면 멈췄던 함수가 계속 실행됨

yield '반환된 value 값';

console.log('이어서 실행');

}

// 함수 코드가 바로 실행되는 대신 이터레이터로 사용할 수있는 Generator 객체를 반환함

let iterator = method();

// 함수가 실행되고, yield를 만나 실행이 멈추게됨

let yieldValue = iterator.next().value;

console.log('yieldValue : ', yieldValue);

// yieldValue : '반환된 value 값'

// 멈췄던 함수의 나머지 부분이 계속 실행됨

iterator.next();

// --> 데이터 스트림을 만들때 유용함

// next에 인자를 전달하면 yield 자리에 전달된다.

function\* foo() {

console.log(yield); // 1

console.log(yield); // 2

console.log(yield); // 3

}

let g = foo();

g.next();

g.next(1);

g.next(2);

g.next(3);

//-------------

// 루프 만드는 방법 예제

function\* loop(name){

while (true){

yield Math.random() \* 100;

console.log(`name : ${name}`);

}

}

let generator = loop('루프 연습');

const max = 30;

let val = 100;

while (val > max){

val = generator.next().value;

console.log(`반환된 값 : ${val}`);

}

console.log(`종료전 마지막 값 : ${val}`);

/\* 실행

반환된 값 : 51.66957093797133

name : 루프 연습

반환된 값 : 78.49644577554879

name : 루프 연습

반환된 값 : 24.601733854516805

종료전 마지막 값 : 24.601733854516805

\*/

//---------------------------------------

// 비구조화 (Destructureing)

//---------------------------------------

// 비구조화 표현식 : 패턴 등호(=) 분해 대상 객체 (객체, 배열)

//-------------

// 객체 분해

function getMember(){

return {symbol:'출력', price: 10};

}

// 패턴 이외의 다른 프로퍼티가 있어도 패턴 이외의 속성은 무시됨

let {symbol, price, other} = getMember();

console.log(`${symbol} : ${price}, ${other}`);

// other 속성은 추출할 값이 없으므로 undefined

// 출력 : 10, undefined

// 비구조화 할당에도 기본값을 지정할 수 있음

let {symbol, price, other='기본값'} = getMember();

console.log(`${symbol} : ${price}, ${other}`);

// 출력 : 10, 기본값

// 새로운 변수에 할당하고 싶으면 명시적으로 이름을 할당한다.

let {price: newName, symbol: anotherName} = getMember();

// 새로운 newName, anotherName 변수명을 사용할 수 있음 (symbol, price 변수명은 정의 안됨)

console.log(`${anotherName} : ${newName}`);

// 출력 : 10

let {price: newName, symbol: anotherName='기본값'} = getMember();

// 이경우 기본값은 anotherName 변수에 한해 적용된다. (symbol에는 적용 안됨)

// 중첩된 객체에도 같은 방식으로 할당이 적용된다.

function getMember(){

return {

symbol:'출력',

price: {a: 'a값', b: 'b'}

};

}

let {symbol, price: {a:newA}} = getMember();

console.log(`${symbol} : ${newA}`);

// 출력 : a값

console.log(`${symbol} : ${price}`); // 에러 : price변수 찾을 수 없음

console.log(`${symbol} : ${a}`); // 에러 : a변수 찾을 수 없음

//-------------

// 배열 분해

// 배열에는 변수 이름이 없으므로 인덱스 순서대로 할당됨

let [name1, name2] = [1, 2];

console.log(`name1=${name1}, name2=${name2}`);

// name1=1, name2=2

// 두번째 항목만 추출하고 싶을때

let [, name2] = [1, 2];

// 나머지 연산자와 함께 사용

let [name1, name2, ...rest] = [1, 2, 3, 4, 5];

// rest : [3,4,5]

console.log(`rest=${rest}`);

// rest=3,4,5

// 이 방식은 함수를 선언할때도 사용할 수 있음

function arrayParam([name1, name2, ...rest]) {

console.log(`rest=${rest}`);

}

arrayParam([1, 2, 3, 4, 5]);

// rest=3,4,5

//---------------------------------------

// 반복문

//---------------------------------------

var ar = [1,2,3];

ar.prop1 = '변수';

// 컬렉션 데이터만 순회

ar.forEach(n => console.log(n));

// prop1 은 순환하지 않는다.

// 중간에 멈출수 없다.

// 중간에 멈출려면 every() 함수를 사용한다.

// 컬렉션 데이터와 프로퍼티 모두 순회 (key을 기준으로)

for(let n in ar){

console.log(n, ar[n])

}

// prop1 까지 순환하여 탐색한다.

// 컬렉션 데이터만 순회 (value를 기준으로)

for(let n of ar){

console.log(n)

}

// prop1 은 순환하지 않는다.

// 문자열, Array, Map, Set 같은 반복 가능한 객체에 모두 사용 가능.

//---------------------------------------

// 모듈

//---------------------------------------

// import : 다른 파일에 있는 변수나 함수를 불러와 사용

// export : 모듈에 있는 변수, 함수, 클래스를 외부로 공개할 수 있다.

// tax.js

export var taxCode;

export function calcTaxes(){}

function doSomething() {}

// 외부로 공개되는 항목중 하나(클래스, 변수, 함수)는 기본 항목으로 지정할 수 있음

export default function () {console.log('default function');}

// 사용하는 다른 파일 ('tax.js'에서 확장자 생략 가능)

import {taxCode, calcTaxes} from 'tax';

// 기본 항목으로 지정된 함수는 새 이름을 붙여 사용

import funcName, {taxCode} from 'tax';

funcName();

// 출력 : default function

// 이미 지정된 항목에 새로운 이름을 붙여 사용 : as 키워드

import funcName, {taxCode as tax2016} from 'tax';

// tax2016 이름으로 사용 가능함

//---------------------------------------

// 객체 리터럴 프로퍼티 기능 확장

//---------------------------------------

// 프로퍼티 이름을 생략(Property shorthand)

let x = 1, y = 2;

const obj = { x, y }; // { x: 1, y: 2 }

// 프로퍼티 이름을 동적으로 생성(Computed property name)

const propNamePrefix = 'prop\_';

const obj = {

[propNamePrefix + ++i]: i,

[propNamePrefix + ++i]: i,

[propNamePrefix + ++i]: i

};

// { prop\_1: 1, prop\_2: 2, prop\_3: 3 }

// 메소드 축약 표현

const obj = {

name: 'Lee',

sayHi() {

console.log('Hi! ' + this.name);

}

};

obj.sayHi(); // Hi! Lee

**데이터 타입**

// http://www.typescriptlang.org/docs/handbook/basic-types.html

//---------------------------------------

// Intersection Types

// 기존 유형을 모두 추가하여 필요한 모든 기능을 갖춘 단일 유형을 얻을 수 있음

function extend<T, U>(first: T, second: U): T & U {

let result = <T & U>{};

return result;

}

//---------------------------------------

// Union Types

interface Bird {

fly();

layEggs();

}

interface Fish {

swim();

layEggs();

}

function getSmallPet(): Fish | Bird {

let result:Fish | Bird;

//...

return result;

}

let pet = getSmallPet();

pet.layEggs(); // okay

// pet.swim(); // errors

// 동일한 코드가 작동하도록 하려면 type assertion을 사용해야함.

((<Fish>pet).swim) ? (<Fish>pet).swim() : (<Bird>pet).fly();

// type assertion을 여러번 사용 했음.

// 다음과 같이 type guard를 사용할 수도 있음

function isFish(pet: Fish | Bird): pet is Fish {

return (<Fish>pet).swim !== undefined;

}

(isFish(pet)) ? pet.swim() : pet.fly();

// 응용

function isNumber(x: any): x is number { return typeof x === "number"; }

function isString(x: any): x is string { return typeof x === "string"; }

//---------------------------------------

// Array

let list: number[] = [1, 2, 3];

let list1: Array<number> = [1, 2, 3];

//---------------------------------------

// Tuple

// 정의. 정의되지 않은 index는 union Type 으로 처리됨 (string | number)

let x: [string, number];

// OK

x = ["hello", 10];

// Error

// x = [10, "hello"];

// OK

console.log(x[0].substr(1));

// Error, 'number' does not have 'substr'

// console.log(x[1].substr(1));

// OK, 'string' can be assigned to 'string | number'

x[3] = "world";

// OK, 'string' and 'number' both have 'toString'

console.log(x[5].toString());

// Error, 'boolean' isn't 'string | number'

// x[6] = true;

//---------------------------------------

// Enum

// http://www.typescriptlang.org/docs/handbook/enums.html

// 기봄적으로 0부터 카운팅

enum Color {Red, Green, Blue}

let c: Color = Color.Green;

// 명시적으로 1부터 카운팅

enum Color1 {Red = 1, Green, Blue}

let d: Color1 = Color1.Green;

// 모든 멤버를 지정할 수도 있다.

enum Color2 {Red = 1, Green = 2, Blue = 4}

let e: Color2 = Color2.Green;

// 열거 형의 편리한 기능은 숫자 값에서 열거 형의 값의 이름으로 이동할 수 있다

// 예를 들어 값이 2있지만 Color위 의 enum에 매핑 된 값이 확실하지 않은 경우

// 해당 이름을 찾을 수 있다.

enum Color3 {Red = 1, Green, Blue}

let colorName: string = Color3[2];

// Green

alert(colorName);

enum Enum { A }

let a = Enum.A;

let nameOfA = Enum[a]; // "A"

const enum Enum1 {

A = 1,

B = A \* 2

}

//---------------------------------------

// Any

let notSure: any = 4;

notSure.ifItExists(); // okay, ifItExists might exist at runtime

notSure.toFixed(); // okay, toFixed exists (but the compiler doesn't check)

// Object 유형과 차이점

// let prettySure: Object = 4;

// prettySure.toFixed(); // Error: Property 'toFixed' doesn't exist on type 'Object'.

// 배열에 서로 다른 유형의 값이 혼재할때

let list2: any[] = [1, true, "free"];

list2[1] = 100;

//---------------------------------------

// Void

// undefined, null 값만 할당 가능

let unusable: void = undefined;

// 주로 함수의 결과값을 반환하지 않는 경우에 사용

//---------------------------------------

// Null, Undefined

let u: undefined = undefined;

let n: null = null;

// 기본적으로 null및 undefined다른 모든 유형의 하위 유형

// (다른 유형의 값으로 할당 가능)

// --strictNullChecks 플래그를 사용하면

// 각 유형과 void 타입에만 할당할 수있음

// 문자열또는 null, undefined 를 전달하려면 union 타입(string | null | undefined)을 명시적으로 사용

function f(x: number, y?: number) {

return x + (y || 0);

}

f(1, undefined);

f(1, null); // error, 'null' is not assignable to 'number | undefined'

//---------------------------------------

// Never

// 결코 발생되지 않는 유형을 표현 (error exception)

// 함수 표현식 또는 화살표 함수 표현식에서 반환값으로 사용

// never 유형만 할당 가능 (any 타입도 안됨)

// Function returning never must have unreachable end point

function error(message: string): never {

throw new Error(message);

}

// Inferred return type is never

function fail() {

return error("Something failed");

}

// Function returning never must have unreachable end point

function infiniteLoop(): never {

while (true) {}

}

//---------------------------------------

// Type assertions (type cast)

//---------------------------------------

// 명시적으로 타입을 알고 있을때 사용

// 방법1. 꺽쇠 괄호 구문

let someValue: any = "this is a string";

let strLength: number = (<string>someValue).length;

// 방법2. as syntax

let someValue1: any = "this is a string";

let strLength1: number = (someValue1 as string).length;

//---------------------------------------

// Generics

// http://www.typescriptlang.org/docs/handbook/generics.html

//---------------------------------------

// 유형 변수(type variable) T 를 사용하여 반환되는 값의 유형 정보를 캡쳐

function identity <T>(arg: T): T {

return arg;

}

// 호출 방식 1 : argument로 유형(string)을 <>로 함께 전달하여 호출함.

let output = identity<string>("myString"); // type of output will be 'string'

// 호출 방식 2 : 타입 인자 추론 (자동으로 유형을 추론)

let output1 = identity("myString"); // type of output will be 'string'

// arg의 길이를 출력하고 싶을때

// arg.length ( Error: T doesn't have .length )

// arg를 배열로 취급되도록 재정의하면

function loggingIdentity<T>(arg: T[]): T[] {

console.log(arg.length); // Array has a .length, so no more error

return arg;

}

function loggingIdentity1<T>(arg: Array<T>): Array<T> {

console.log(arg.length); // Array has a .length, so no more error

return arg;

}

//---------------------------------------

// 제네릭 함수의 유형 표현

// 방식 1 : <T>(arg: T) => T

let myIdentity: <T>(arg: T) => T = identity;

// 방식 2 : {<T>(arg: T): T}

// 객체 리터럴(object literal) 방식으로 유형 표현

let myIdentity1: {<T>(arg: T): T} = identity;

// 제네릭 인터페이스

interface GenericIdentityFn {

<T>(arg: T): T;

}

let myIdentity2: GenericIdentityFn = identity;

// 이방식은 type 매개변수까지 노출할 수 있다.

interface GenericIdentityFn1<T> {

(arg: T): T;

}

let myIdentity3: GenericIdentityFn1<number> = identity;

// 제네릭 클래스

// 정적 클래스 멤버에는 T 사용할 수 없음

class GenericNumber<T> {

zeroValue: T;

add: (x: T, y: T) => T;

}

let myGenericNumber = new GenericNumber<number>();

myGenericNumber.zeroValue = 0;

myGenericNumber.add = function(x, y) { return x + y; };

// 일반 열거형, 네임 스페이스는 만들 수 없음.

//---------------------------------------

// Generic Constraints (제약 조건)

// 위에서 identity 함수의 args는 .length 속성에 대해 에러가 발생 했음

// extends 키워드를 통해 인터페이스를 상속하여 조건을 명시할 수 있음

interface Lengthwise {

length: number;

}

function logIdentity<T extends Lengthwise>(arg: T): T {

console.log(arg.length); // Now we know it has a .length property, so no more error

return arg;

}

// Error, number doesn't have a .length property

// logIdentity(3);

// 인터페이스를 만족하는 유형의 값을 전달해야함

logIdentity({length: 10, value: 3});

// 제약 조건에서 사용하기

function getProperty<T, K extends keyof T>(obj: T, key: K) {

return obj[key];

}

let y = { a: 1, b: 2, c: 3, d: 4 };

getProperty(y, "a"); // 1

// error: Argument of type 'm' isn't assignable to 'a' | 'b' | 'c' | 'd'.

// getProperty(y, "m");

//---------------------------------------

// 제네릭에서 클래스 유형 사용 (팩토리를 생성)

// 예제 1 : 생성자 함수를 사용하여 클래스 유형을 참조

function create<T>(c: {new(): T; }): T {

return new c();

}

// 예제 2 : 상속 고급 예제

class BeeKeeper { hasMask: boolean; }

class ZooKeeper { nametag: string; }

class Animal { numLegs: number; }

class Bee extends Animal { keeper: BeeKeeper; }

class Lion extends Animal { keeper: ZooKeeper; }

function createInstance<A extends Animal>(c: new () => A): A {

return new c();

}

createInstance(Lion).keeper.nametag; // typechecks!

createInstance(Bee).keeper.hasMask; // typechecks!

//---------------------------------------

// Comparing two functions

//---------------------------------------

(()=>{

let x = (a: number) => 0;

let y = (b: number, s: string) => 0;

y = x; // OK

// x = y; // Error

})();

(()=>{

let items = [1, 2, 3];

// Don't force these extra parameters

items.forEach((item, index, array) => console.log(item));

// Should be OK!

items.forEach(item => console.log(item));

})();

(()=>{

let x = () => ({name: "Alice"});

let y = () => ({name: "Alice", location: "Seattle"});

x = y; // OK

// y = x; // Error because x() lacks a location property

})();

(()=>{

// 열거 형은 숫자와 호환, 숫자는 열거 형과 호환,

// 다른 열거 형에서 가져온 Enum 값은 호환되지 않음

enum Status { Ready, Waiting };

enum Color { Red, Blue, Green };

let status = Status.Ready;

// status = Color.Green; //error

})();

(()=>{

// 클래스 유형의 두 객체를 비교할 때 인스턴스의 멤버 만 비교됨

class Animal {

feet: number;

constructor(name: string, numFeet: number) { }

}

class Size {

feet: number;

constructor(numFeet: number) { }

}

let a: Animal;

let s: Size;

a = s; //OK

s = a; //OK

})();

(()=> {

interface Empty<T> {}

let x: Empty<number>;

let y: Empty<string>;

x = y; // okay, y matches structure of x

})();

(()=> {

interface NotEmpty<T> {

data: T;

}

let x: NotEmpty<number>;

let y: NotEmpty<string>;

// x = y; // error, x and y are not compatible

})();

**TypeScript**

// 인자 기본값 (함수도 사용 가능)

function method (a:any, b:string='b'):number{ return 0; }

function method1 (a:any, b:string=defaultValue()):number{ return 0; }

function defaultValue():string{ return 'default'; }

// defaultValue 함수는 method1 함수가 실행될 때마다 실행되므로 성능에 신경 써야함

// 옵션 인자

function method2 (a:any, b:string='b', c?:string):number{ return 0; }

// 화살표 함수 (람다 표현식)

let getName = ()=> '이름'; // 1 줄일때 return 생략 가능

let getName1 = ()=> { return '이름'; }; // 여러 줄일때 {} 블럭 사용

setInterval(()=>{ return this; }, 1000); // 블럭 내부에 this context 유지됨

// 함수 오버로딩 (변환되면서 합쳐지기 때문에 완벽한 기능은 아님)

function attr (name:string):string;

function attr (name:string, value:string):void;

function attr (map:any):void;

// --> 변환되는 코드는 다음과 같이 하나의 함수로 합쳐짐

function attr(nameOrMap: any, value?:string): any{

if(nameOrMap && typeof nameOrMap === 'string'){

// string 타입 인자가 전달된 경우

}else{

// any 타입 인자가 전달된 경우

}

// value 인자 처리

}

//---------------------------------------

// 클래스

//---------------------------------------

// 클래스 선언은 호이스팅 되지 않는다. 따라서 스크립트에서 사용되기 전에 선언해야 한다.

class Person{

// 생성자에서 접근제한자를 지정할 수도 있음

constructor(private \_ssn?: string){

this.\_ssn = \_ssn;

}

// 접근자를 생략하면 기본적으로 public 으로 인식

age:number;

// 접근 제한자 (public, protected, private)

// private \_ssn:string;

get ssn():string{

return this.\_ssn;

}

set ssn(value: string){

this.\_ssn = value;

}

method (a: any):void{}

// 정적 클래스 멤버

static method1 (a: any):void{}

// 참고 : ES6는 클래스 멤버 변수를 지원하지 않는다.

// class Person{ var prop; } (X)

// 정적 변수는 클래스 선언 밖에서 정의하여 사용할 수 있다.

// class Person{}

// Person.counter = 0;

}

// 클래스 멤버 사용

let p = new Person('ssn');

p.age = 5;

p.method(5);

// 정적 클래스 멤버 사용

Person.method1(5);

//---------------------------------------

// 클래스 상속

//---------------------------------------

// extends 키워드가 사용되면 tsc 변환 과정에서 class 상속이 prototype 상속으로 변환된다.

class Employee extends Person{

department: string;

// 생성자를 재정의할때 반드시 super 를 호출해야 함

constructor(\_ssn?: string){

super(\_ssn);

}

method (a: any):void{

super.method(a);

}

}

//---------------------------------------

// Generics :

//---------------------------------------

// - 실행 시점에 명확한 타입을 지정할때 사용

// - 배열의 항목을 특정 타입으로 제한할때 사용

let workers: Array<Person> = [];

function something<T> (data: T) {}

something<string>('Hello');

// something<string>(123); <-- 컴파일 error 발생 (런타임에는 에러 발생 안함)

//---------------------------------------

// 인터페이스

//---------------------------------------

// - 커스텀 타입으로 사용

// - 추상 클래스로 사용

interface IPlayable{

age?:number;

play(percent:number):boolean;

}

// 커스텀 타입으로 사용

class Person1{

constructor(config: IPlayable){}

}

// interface 상속 여부와 상관 없이 객체의 형태가 같으면 호환되는 것으로 판단한다.

let config: IPlayable = {

age: 5,

play: function(){

return true;

}

};

let p1 = new Person1(config);

// 추상 클래스로 사용 (여러개의 인터페이스 사용 가능함)

class Employee1 extends Person1 implements IPlayable{

age?:number = 50;

play(percent:number):boolean{

return true;

}

}

let workers1: Array<IPlayable> = [];

workers1[0] = new Employee1(config);

workers1.forEach((work)=>work.play(work.age));

// 실행할 수 있는 인터페이스

// - 인터페이스 메서드 정의할때 익명함수를 사용하면 callable interface를 정의할 수 있다.

interface ICallable{

(percent:number): boolean;

}

class ClassA{

constructor (private validator: ICallable){

//validator(100);

}

}

const callableA: ICallable = (percent:number) => { return true; };

const callableB: ICallable = (percent:number) => { return false; };

const callableWorker:Array<ClassA> = [];

callableWorker[0] = new ClassA(callableA);

callableWorker[1] = new ClassA(callableB);

// typescript는 클래스도 하나의 인터페이스로 간주한다.

class ClassC {}

// extends 키워드가 아닌 implements 키워드를 사용한다.

class ClassD implements ClassC {}

//---------------------------------------

// Polymorphic this types (다형성)

//---------------------------------------

class BasicCalculator {

public constructor(protected value: number = 0) { }

public currentValue(): number {

return this.value;

}

public add(operand: number): this {

this.value += operand;

return this;

}

public multiply(operand: number): this {

this.value \*= operand;

return this;

}

// ... other operations go here ...

}

let v = new BasicCalculator(2)

.multiply(5)

.add(1)

.currentValue();

//---------------------------------------

// Type Aliases (유형 별칭)

//---------------------------------------

// 실제 새 유형을 작성하는 것이 아님. 별칭일 뿐.

(()=>{

type Name = string;

type NameResolver = () => string;

type NameOrResolver = Name | NameResolver;

function getName(n: NameOrResolver): Name {

return (typeof n === "string") ? n : n();

}

type Container<T> = { value: T };

// 속성에서 유형 별칭을 참조 할 수 있음

type Tree<T> = {

value: T;

left: Tree<T>;

right: Tree<T>;

}

// Intersection Types 함께 사용

type LinkedList<T> = T & { next: LinkedList<T> };

interface Person { name: string; }

var people: LinkedList<Person>;

var s = people.name;

var s = people.next.name;

var s = people.next.next.name;

var s = people.next.next.next.name;

// 선언문의 오른쪽에는 사용할 수 없음

// type Yikes = Array<Yikes>; // error

})();

// 인터페이스와 차이점

// - 오류 메시지는 별칭 이름을 사용하지 않음

// - 유형 별칭을 확장하거나 구현할 수 없으며 다른 유형을 확장 / 구현할 수도 없다

// - 인터페이스로 표현할 수없는 Union 이나 Tuple type에 사용

type Alias = { num: number }

interface Interface { num: number; }

declare function aliased(arg: Alias): Alias;

declare function interfaced(arg: Interface): Interface;

// 문자열 리터럴 유형

(()=>{

type Easing = "ease-in" | "ease-out" | "ease-in-out";

let easing: Easing = "ease-in";

// error: Argument of type '"uneasy"' is not assignable to parameter of type '"ease-in" | "ease-out" | "ease-in-out"'

// easing = "uneasy";

})();

// 오버로드를 구별하기 위해 동일한 방법으로 문자열 리터럴 유형을 사용할 수 있음

function createElement(tagName: "img"): HTMLImageElement;

function createElement(tagName: "input"): HTMLInputElement;

function createElement(tagName: string): Element{

let a:Element;

return a;

}

// tagged unions or algebraic data types

// combine string literal types, union types, type guards, and type aliases

interface Square { kind: "square"; size: number; }

interface Rectangle { kind: "rectangle"; width: number; height: number; }

interface Circle { kind: "circle"; radius: number; }

(()=>{

// kind 문자열 리터럴 형식을 확인

type Shape = Square | Rectangle | Circle;

function area(s: Shape) {

switch (s.kind) {

case "square": return s.size \* s.size;

case "rectangle": return s.height \* s.width;

case "circle": return Math.PI \* s.radius \*\* 2;

}

}

})();

/\*

interface Triangle { height: number; }

(()=>{

// Triangle의 kind를 인식할 수 없음

type Shape = Square | Rectangle | Circle | Triangle;

function area(s: Shape) {

switch (s.kind) {

case "square": return s.size \* s.size;

case "rectangle": return s.height \* s.width;

case "circle": return Math.PI \* s.radius \*\* 2;

default: return assertNever(s); // error here if there are missing cases

}

// should error here - we didn't handle case "triangle"

}

function assertNever(x: never): never {

throw new Error("Unexpected object: " + x);

}

})();

//\*/

//---------------------------------------

// Symbols

// http://www.typescriptlang.org/docs/handbook/symbols.html

//---------------------------------------

((Symbol)=>{

// number, string 과 같은 primitive 데이터 타입임.

// constructor로 생성하여 사용

let sym1 = Symbol();

let sym2 = Symbol("key"); // optional string key

let sym3 = Symbol("key");

sym2 === sym3; // false, symbols are unique

// string 과 같이 object의 key로 사용할 수 있음

let obj = { [sym1]: "value" };

console.log(obj[sym1]); // "value"

// 클래스와 결합하여 사용

class C {

[sym2](){

return "C";

}

}

let c = new C();

let className = c[sym2](); // "C"

})();