TypeScript

1. What is TypeScript?

1.1 What is TypeScirpt?

- ✓ TypeScript는 안전하고 예측 가능한 코드 구현에 초점을 맞춘 JavaScript의 상위 집합(Superset) 언어 입니다.
- ✓ TypeScript의 가장 큰 특징은 기존 JavaScript에 Type System을 적용한 것입니다.
- ✓ TypeScript로 작성된 코드는 TypeScript 컴파일러(tsc)를 통해 JavaScript 변환되고 이후 실행됩니다.
- ✓ Type System 이외에도 TypeScript는 Decorators와 같은 다양한 기능을 제공합니다.

TypeScript = JavaScript + **Type System**

JavaScript

```
function add(n1, n2) {
  return n1 + n2;
}

console.log(add('1', '2')); // 12
```

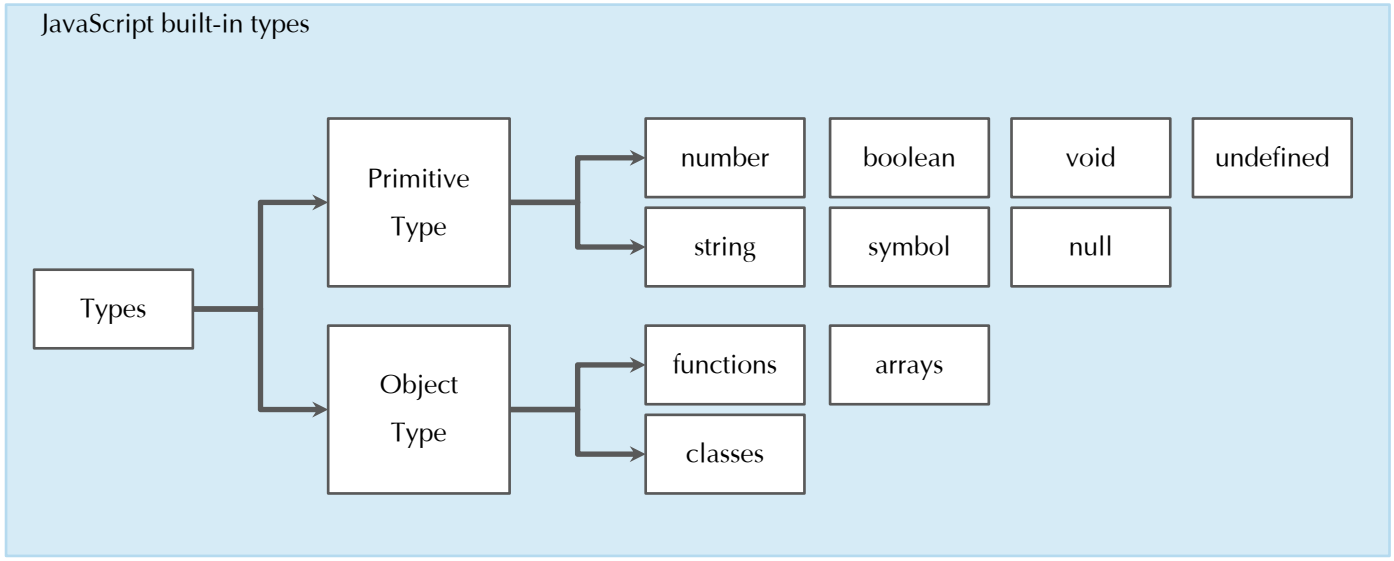
TypeScript

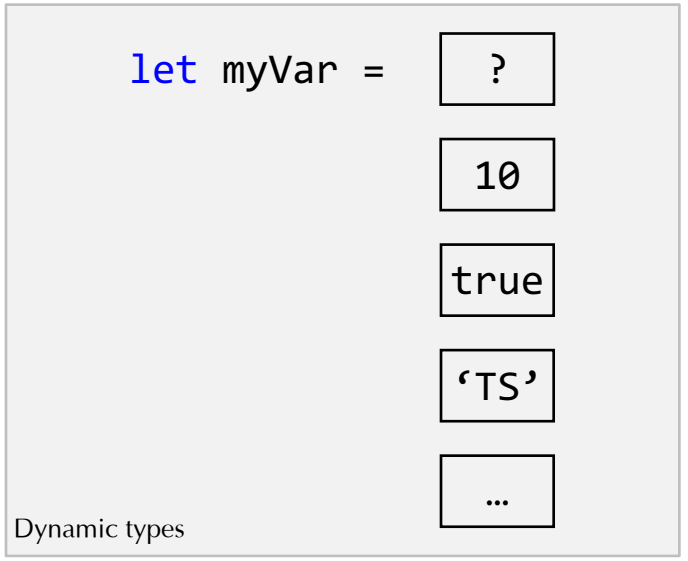
```
function add(n1: number, n2: number): number {
  return n1 + n2;
}

console.log(add('1', '2')); // compile error
```

```
Argument of type '"1"' is not assignable to parameter of type 'number'
```

1.2 Type System(1/2)

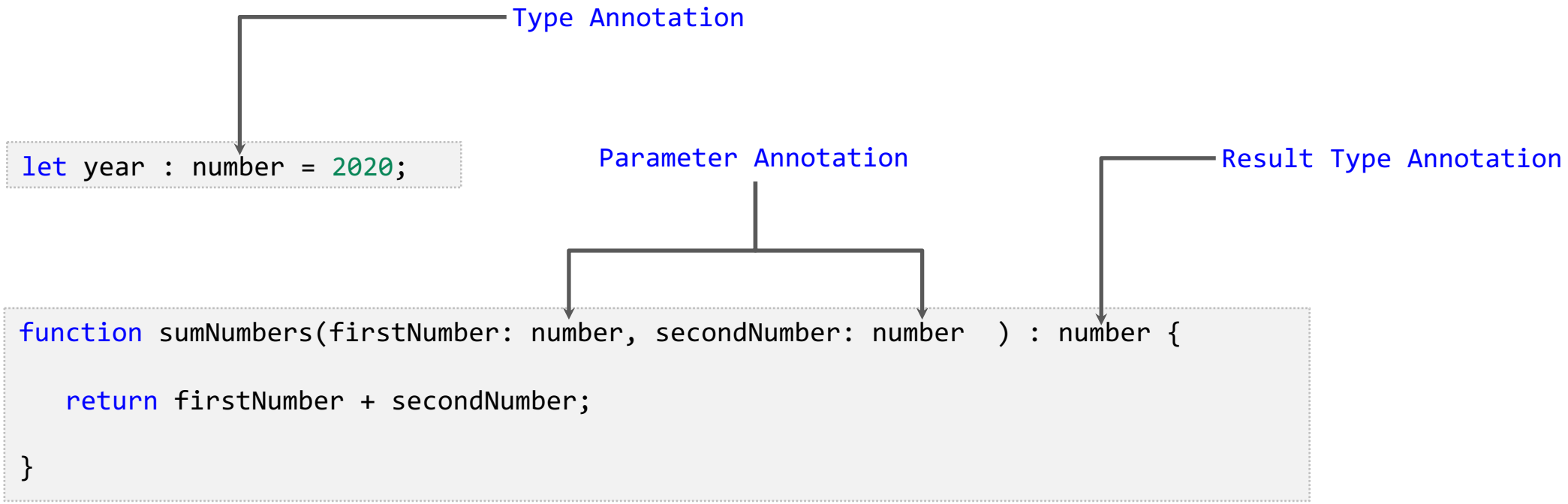
- ✓ TypeScript의 가장 큰 특징은 JavaScript의 타입 체계에 정적 타입(static typed)을 적용할 수 있도록 한 것입니다.
- ✓ JavaScipt는 동적 타입 체계(dynamic typed)의 언어로 변수에 타입을 지정하지 않습니다.
- ✓ 동적 타입 체계는 프로그램에 유연성을 제공하지만 변수에 할당된 값을 읽어 사용하는 단계에서 문제가 발생할 수 있습니다.
- ✓ TypeScript는 변수, 파라미터에 타입을 지정할 수 있으며 선언된 타입의 값이 아닌 경우 에러를 발생합니다.





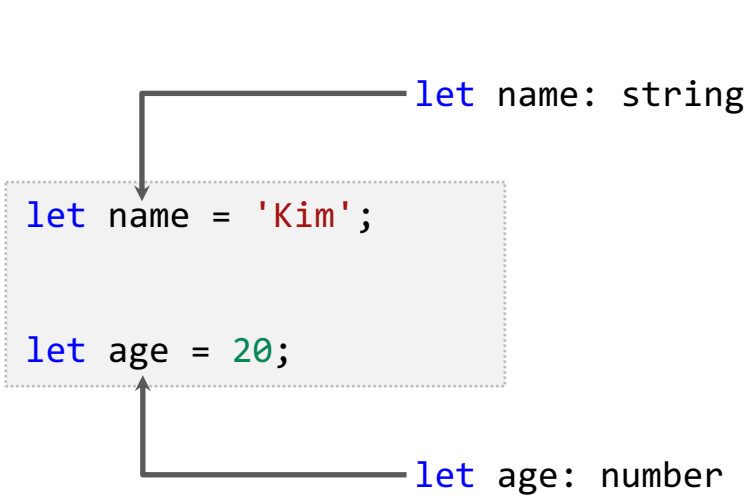
1.2 Type System(2/2)

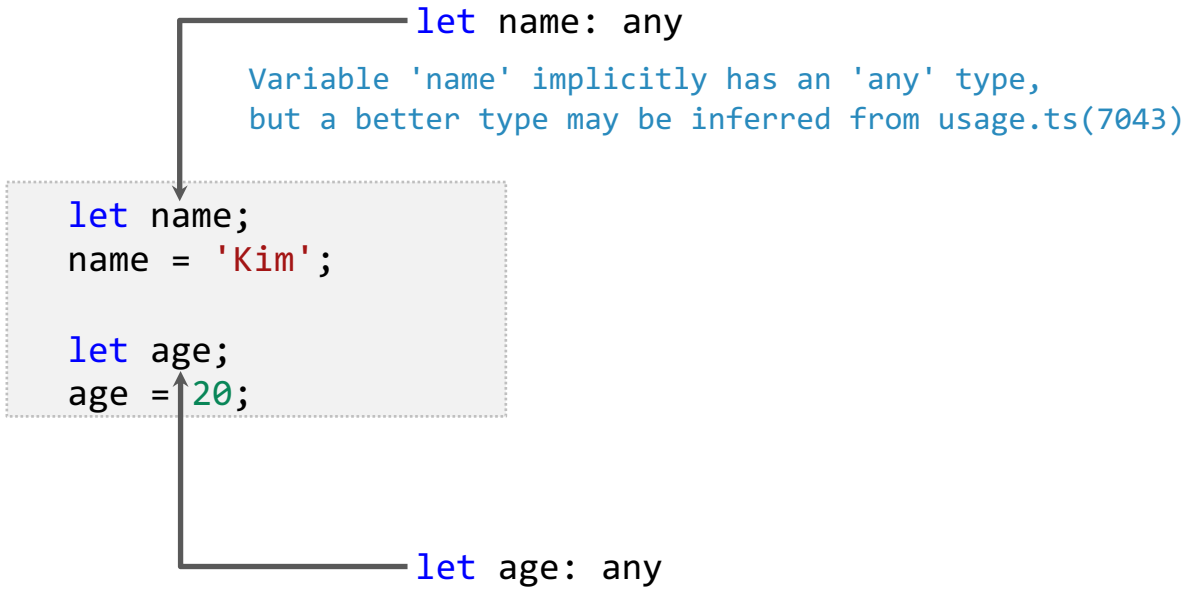
- ✓ TypeScript는 정적 타입을 지원합니다. 변수의 정의와 함께 대입할 값의 타입을 함께 지정합니다.
- ✓ 변수를 정의하면서 타입을 함께 지정하면 해당 타입의 데이터만 대입 가능하며 다른 타입의 값을 대입하면 컴파일 시점에 에러가 발생합니다.
- ✓ 변수의 선언과 마찬가지로 함수의 매개변수, 반환값의 정의에도 동일한 시스템을 적용할 수 있습니다.
- ✓ 매개변수에 타입을 지정하면 해당 타입의 데이터를 전달 인자로 하며 반환 값도 동일합니다.



1.3 Type Inference

- ✓ 정적으로 타입을 명시하지 않고 대입(assign)하는 값을 통해 타입을 유추해 결정하는 것이 타입 추론 입니다.
- ✓ 기본적으로 변수의 선언과 함께 값을 대입하는 경우 해당 변수는 타입 추론으로 인해 대입되는 값의 타입을 갖습니다.
- ✓ 변수를 선언하고 같은 코드 라인에서 특정 값을 대입하면 TypeScript는 대입값의 타입을 유추해 변수의 타입을 확정합니다.
- ✓ 변수의 선언과 값의 대입을 다른 코드 라인에서 진행하면 해당 변수는 any 타입의 변수가 됩니다.





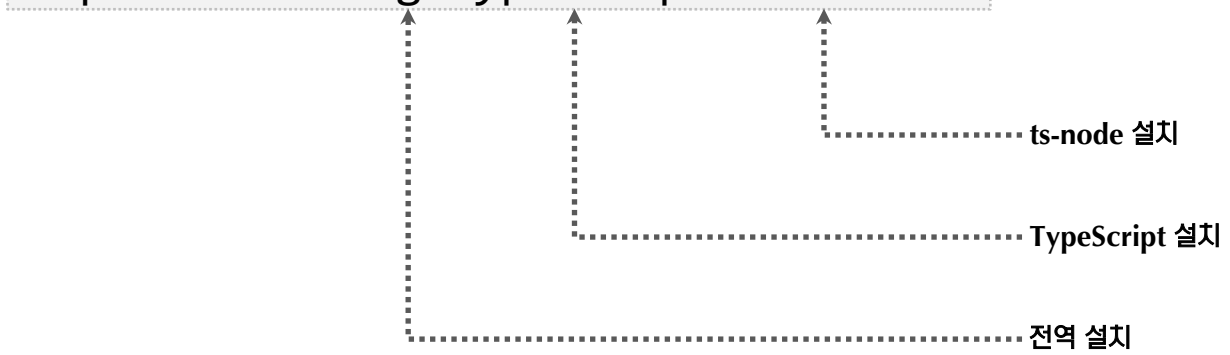
2.Step#01 - Annotations

2.1 개발 환경 구성
2.2 Todo List 개발 - Defining the Data Model
2.3 Type Annotations

2.1 개발환경 구성(1/2)

- ✓ TypeScript 개발 환경 구성을 위한 필수 모듈은 typescript.js 입니다.
- ✓ node 설치를 기본 전제로 npm(혹은 yarn)을 이용한 typescript를 설치하면 tsc를 이용해 *.ts 파일에 대한 컴파일이 가능합니다. (ts-node를 설치하면 컴파일과 실행을 한번에 수행할 수 있습니다.)
- ✓ TypeScript로 작성한 *.ts 파일은 tsc 컴파일 과정을 거치면 *.js 파일로 트랜스컴파일(transcompile) 됩니다.
- ✓ 개발을 위한 IDE는 Visual Studio Code, IntelliJ 등 다양하며 선택해 설치합니다.

```
>npm install -g typescript ts-node
```



2.1 개발환경 구성(2/2)

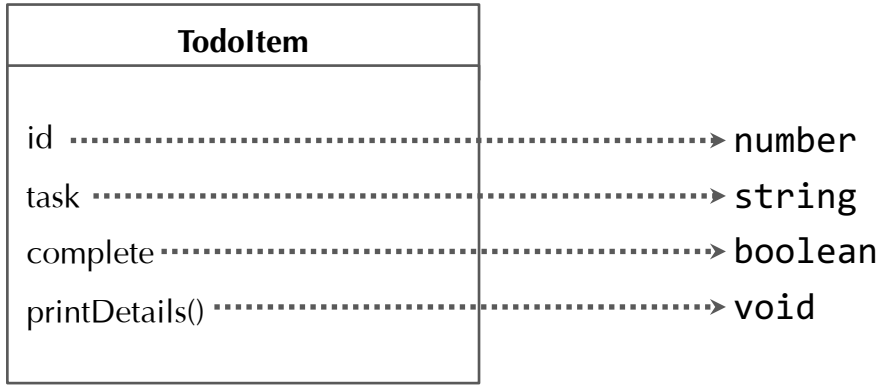
- ✓ 기본적인 프로젝트의 구성 디렉토리는 src, build로 구분하여 작성 소스(*.ts)와 컴파일된 소스(*.js)를 구분합니다.
- ✓ 디렉토리를 구분해서 소스 파일들을 관리하기 위해 tsconfig.json 파일을 생성하고 관련 내용을 수정합니다.
- ✓ package.json 파일을 생성하고 nodemon과 concurrently 모듈을 설치합니다.
- ✓ package.json 파일에 scripts 내용을 추가하여 빌드(build)와 실행(run) 명령을 스크립트로 실행되도록 설정합니다.

1	>tsc --init	tsconfig.json 파일 생성
2-1	"outDir" : "./build"	tsconfig.json 수정 : out 디렉토리 지정
2-2	"rootDir" : "./src"	tsconfig.json 수정 : root 디렉토리 지정
3	>npm init -y	package.json 파일 생성
4	>npm i nodemon concurrently	nodemon과 concurrently 설치
5-1	"start : build" : "tsc -w"	package.json 수정 : build script 명령 추가
5-2	"start : run" : "nodemon build/index.js"	package.json 수정 : run script 명령 추가
5-3	"start" : "concurrently npm:start:*"	package.json 수정 : 빌드와 실행 명령 추가
6	>npm start	start 스크립트 실행(tsc -watch 모드)

2.2 Todo List 개발 – Defining the Data Model

✓ Todo List 프로젝트에서 기본 데이터에 해당하는 TodoItem 클래스를 정의 합니다.

✓ 기본 JavaScript를 이용한 TodoItem과 TypeScript의 TodoItem 클래스의 차이를 확인합니다.



Todo Data

```
[
  { id: 1, task: 'Buy Some Food', complete: true },
  { id: 2, task: 'To study TypeScript', complete: false },
  { id: 3, task: 'Call sister', complete: false }
]
```

실행 결과

```
My Todo List
1       Buy Some Food   (complete)
2       To study TypeScript
3       Call sister
```

2.3 Type Annotations (1/3) – Variables, Object Literal Annotations

✓ 정적 타입을 기본으로 하는 C, Java와 마찬가지로 변수 선언에 타입을 지정할 수 있습니다.
✓ 변수에 대한 타입을 지정하면 해당 변수에 다른 타입의 값을 대입(assign) 할 수 없습니다.
✓ 변수에 정의한 타입과 다른 타입의 값을 대입하는 코드는 TypeScript 컴파일러에 의해 컴파일 오류가 발생됩니다.
✓ 객체 리터럴에 대한 타입 정의는 해당 리터럴 객체가 갖는 각각의 프로퍼티에 대한 타입을 나열하는 형태로 정의합니다.

```ts
// primitive types
let name: string = 'Kim';
let age: number = 20;
let hasName: boolean = true;

let nullValue: null = null;
let nothing: undefined = undefined;

// built in objects
let now: Date = new Date();

// Array
let colors: string[] = ['red', 'yellow', 'blue'];
let numbers: number[] = [1, 2, 3, 4, 5];
```

```ts
// Class
class Person {}
let person: Person = new Person();

// Object literal
let point: { x: number; y: number } = {
  x: 10,
  y: 20,
};
```

2.3 Type Annotations (2/3) – Functions(1/2)

✓ 함수를 정의할 때 파라미터와 반환값에 타입을 지정합니다.
✓ 함수의 종류(선언적 함수, 익명함수, 람다 함수)에 따라 타입 지정에 대한 표현에는 차이가 있습니다.
✓ 함수의 파라미터를 정의할 때 선택적 매개변수(Optional Parameter)를 지정할 수 있습니다.

선언적 함수

```
function add(n1: number, n2: number): number {
  return n1 + n2;
}
```

익명 함수

```
const add = function(n1: number, n2:number): number{
  return n1+n2;
}
```

람다 함수

```
const add:(n1:number, n2:number) => number = (n1:number, n2:number) : number =>{
  return n1 + n2;
}
```

2.3 Type Annotations (3/3) – Functions(2/2)

- ✓ JavaScript는 기본적으로 가변인자를 통한 함수 호출이 가능했지만 TypeScript는 가변인자를 지원하지 않습니다.
- ✓ 대신 TypeScript는 함수의 오버로딩(Function overloading)을 통해 가변인자와 같은 효과를 구현할 수 있습니다.
- ✓ 선택적 매개변수는 함수 호출시 전달인자를 선택적으로 보낼 수 있는 기능의 매개변수 입니다.
- ✓ 이외에 ES6 버전부터 지원하는 기본 매개변수(Default Parameter), 나머지 매개변수(Rest Parameter)가 있습니다.

함수 오버로딩

```
function add(firstParam: string, secondParam: string): string;
function add(firstParam: number, secondParam: number): number;

function add(firstParam: any, secondParam: any): any {
  console.log(firstParam + secondParam);
}

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

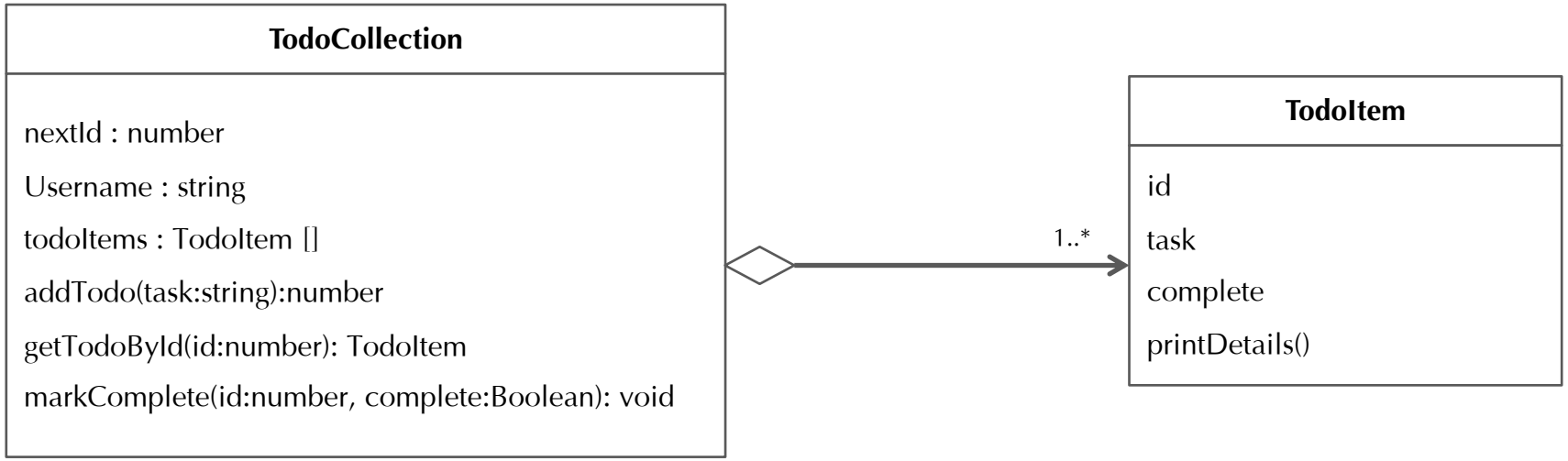
3.Step#02 - Annotations

3.1 Todo List 개발 - Todo List Collection

3.2 Type Annotations

3.1 Todo List 개발 – Todo List Collection

- ✓ TodoItem을 담는 TodoCollection 클래스를 정의합니다.
- ✓ TodoCollection 클래스는 task 추가, task 찾기, task 완료의 기능을 갖습니다.
- ✓ TodoItem은 배열과 맵(Map)에 저장하는 방식 두 가지로 구현합니다.



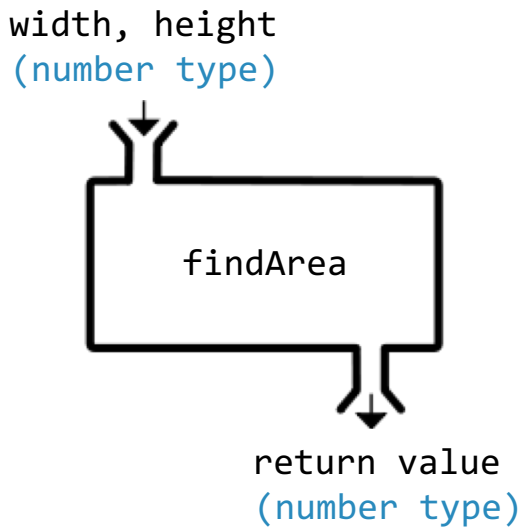
3.2 Type Annotations (1/4) – Inference Around Functions (1/2)

- ✓ 함수의 파라미터를 정의할 때 각 파라미터의 타입을 지정하지 않으면 any 타입의 파라미터가 지정 됩니다.
- ✓ 함수의 반환값에 대한 타입은 return 실행문에 따라 타입 추론(Type Inference)이 적용 됩니다.
- ✓ 파라미터에 타입을 지정하지 않으면 일반 변수와 마찬가지로 암묵적인 any 타입의 적용으로 경고 사항입니다.
- ✓ 반환값의 경우 return 구문으로 명시적인 타입의 유추가 가능합니다.

```
const findArea = function (width: number, heigth: number): number {
  return width * heigth;
}
// const findArea = (width: number, heigth: number): number => {
//   return width * heigth;
// }

console.log(findArea(10, 20)); // 200
```

```
const findArea: (width: number, heigth: number) => number
```

```
const findArea = function (width: number, heigth: number) {
  return width * heigth;
}
```



3.2 Type Annotations (2/4) – Inference Around Functions (2/2)

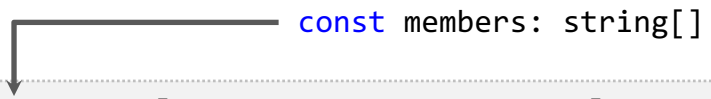
- ✓ 함수의 반환값이 없을 경우 **void** 타입의 반환을 정의 합니다.
- ✓ 함수의 반환값으로 정의 가능한 **never** 타입은 절대 발생하지 않는 값의 타입을 나타냅니다.
- ✓ **void** 타입은 변수로 사용될 경우 **undefined, null** 값만 대입(assign) 가능합니다.
- ✓ **never** 타입은 어떤 타입의 변수에도 대입 될 수 있지만 **never** 타입에는 어떤 타입의 값도 대입될 수 없습니다.

```
let accountBalance: number = 10000;

function overdraftError(message?: string): never {
  throw Error(message);
}

function withdraw(balance: number): number {
  if (accountBalance < balance) {
    overdraftError('잔액이 부족합니다.');
  }
  return accountBalance - balance;
}

console.log(withdraw(3000)); // 7000
console.log(withdraw(15000)); // Error: 잔액이 부족합니다.
```

3.2 Type Annotations (3/4) – Typed Array

- ✓ 배열도 일반 변수와 마찬가지의 타입 시스템이 적용되며 타입의 명시적 지정과 초기화를 통한 타입 지정이 가능합니다.
- ✓ 타입을 명시적 혹은 묵시적으로 지정하지 않을 경우 해당 배열 참조 변수는 any 타입의 변수로 지정됩니다.
- ✓ 배열에 서로 다른 타입의 요소를 저장하는 것이 가능하며 이를 타입으로 지정할 수 있습니다.
- ✓ 서로 다른 타입의 요소로 타입이 지정되면 해당 배열에는 순서에 상관없이 지정된 타입들의 요소를 저장할 수 있습니다.

```
const members : string[] = ['Kim', 'Park', 'Lee'];
```




```
const members = ['Kim', 'Park', 'Lee'];
```

```
const myAry = ['TypeScript', 2020];
// const myAry:(string | number)[] = [];


console.log(myAry[0]);  // TypeScript
console.log(myAry[1]);  // 2020
myAry[2] = 3030;
myAry[3] = 'JavaScript';
```

3.2 Type Annotations (4/4) – Tuples

- ✓ 튜플을 이용하면 배열의 요소 수와 각 요소에 대한 타입을 지정할 수 있습니다.
- ✓ 튜플은 정해진 길이에 맞지 않으면 에러가 발생합니다. 하지만 **push()** 함수를 이용하면 튜플의 규칙은 무시됩니다.
- ✓ 서로 다른 타입의 요소를 갖는 배열은 순서에 상관없이 데이터를 넣을 수 있는 반면 튜플은 정해진 순서에 맞게 넣습니다.
- ✓ 튜플 타입은 배열보다 저장되는 요소에 순서와 수에 제약을 두고자 할 때 사용합니다.

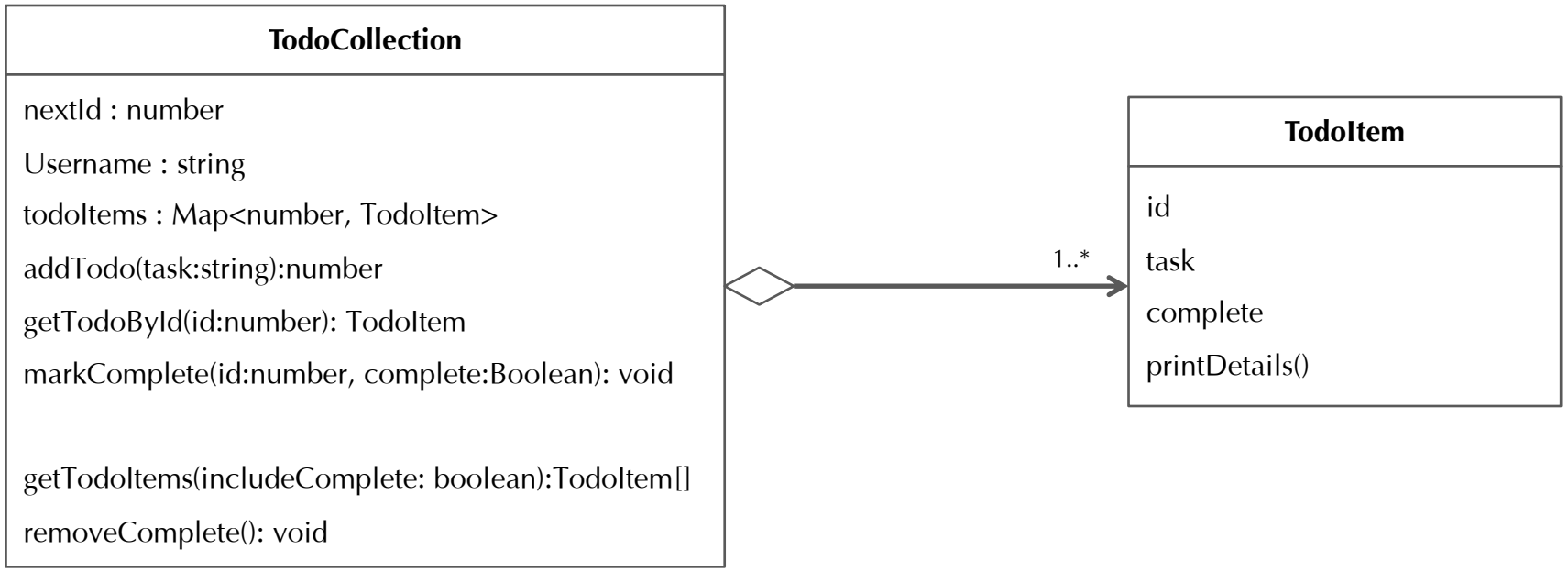
```
const tuples: [string, number] = ['Kim', 30];

tuples[0] = 'Park';   // OK
// tuples[0] = 50;    // Type '50' is not assignable to type 'string'.ts(2322)

tuples[1] = 50;
console.log(tuples);  // [ 'Park', 50 ]

tuples.push(100);
console.log(tuples);  // [ 'Park', 50, 100 ]
```

4.Step#03 - Generic Type

4.1 Todo List 개발 - Map
4.2 Generics
4.3 type alias

4.1 Todo List 개발 – Map

- ✓ TodoItem의 저장을 배열에서 Map 객체로 변경합니다.
- ✓ 전체 할일 목록을 출력하기 위한 메소드를 새로 정의합니다.
- ✓ 할일 목록 중에서 완료된 할일 목록을 삭제하는 메소드를 추가합니다.



4.2 Generics (1/2)

✓ 재사용 가능한 클래스, 함수를 만들기 위해 다양한 타입에서 사용 가능 하도록 하는 것이 제네릭(Generic) 입니다.
✓ 제네릭을 이용하면 모든 타입의 객체를 다루면서 객체 타입의 무결성을 유지할 수 있습니다.
✓ 제네릭을 통해 클래스나 함수 내부에서 사용되는 특정 데이터의 타입을 외부에서 지정합니다.
✓ 제네릭이 적용된 대상(클래스, 함수, 인터페이스)은 선언 시점이 아닌 생성 시점에 사용하는 타입을 결정합니다.

```
class Orange {
  private name = 'Orange';
  constructor(private brix: number = 0) {}
  getName(): string {
    return this.name;
  }
  getBrix(): number {
    return this.brix;
  }
}
```

```
class Apple {
  private name = 'Apple';
  constructor(private brix: number = 0) {}
  getName(): string {
    return this.name;
  }
  getBrix(): number {
    return this.brix;
  }
}
```

```
class Box {
  constructor(private fruit: any = {}) {}
  getFruit(): any {
    return this.fruit;
  }
}
```

```
const box = new Box(new Orange(5));
console.log(box.getFruit().getName()); // Orange
console.log(box);
// Box { fruit: Orange { brix: 5, name: 'Orange' } }
const testBox = new Box('Banana'); // compile pass
console.log(testBox.getFruit().getName());
// runtime error
```

4.2 Generics (2/2)

✓ 제네릭에 사용되는 파라미터는 타입 파라미터(Type Parameter)라 하며 관용적으로 T를 사용합니다.
✓ 제네릭이 적용된 대상은 인스턴스화 될 때 지정된 데이터 타입으로 모든 타입 파라미터의 타입이 지정됩니다.
✓ 타입 파라미터는 상속을 통해 특정 타입의 하위 타입으로 제한 할 수 있습니다.

```
class Orange {
  private name = 'Orange';
  constructor(private brix: number = 0) {}
  getName(): string {
    return this.name;
  }
  getBrix(): number {
    return this.brix;
  }
}
```

```
class Apple {
  private name = 'Apple';
  constructor(private brix: number = 0) {}
  getName(): string {
    return this.name;
  }
  getBrix(): number {
    return this.brix;
  }
}
```

```
class Box<T> {
  constructor(private fruit: T) {}
  getFruit(): T {
    return this.fruit;
  }
}
```

```
const box: Box<Orange> = new Box(new Orange(5));
console.log(box.getFruit().getName()); // Orange
console.log(box);
// Box { fruit: Orange { brix: 5, name: 'Orange' } }
const testBox = new Box('Banana'); // compile pass
console.log(testBox.getFruit().getName());
// compile error
```

4.3 type alias

✓ 새로운 타입을 정의하는 방법은 type alias와 interface를 정의하는 두 가지 방식이 있습니다.

✓ type alias를 이용하면 객체, 공용체(Union), 튜플(Tuple), 기본 타입에 타입의 별칭을 생성할 수 있습니다.

✓ type alias도 제네릭의 사용이 가능하며, 스스로 참조하는 것도 가능합니다.

```
type MyNumber = number;
const n : MyNumber = 10;

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

type User = {name:string, age:number};
const testUser : User = {name:'Kim', age:20};

function printInfo( user: {name:string, age:number}){
  console.log(`User Infomation - Name : ${user.name}, Age : ${user.age}`);
}

function printInfo( user : User){
  console.log(`User Infomation - Name : ${user.name}, Age : ${user.age}`);
}
```

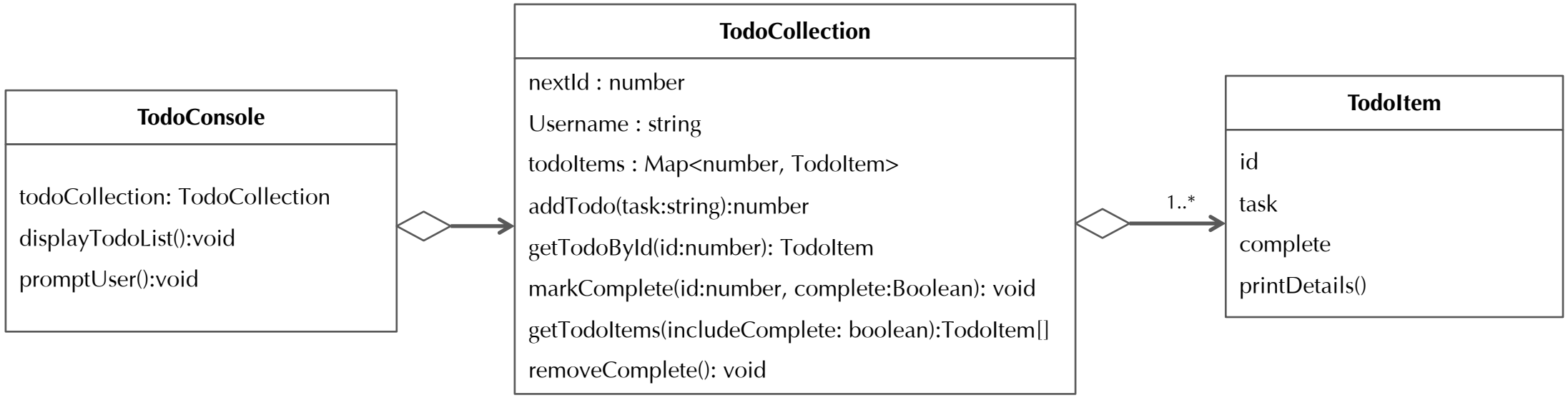
5.Step#05 - User Interaction(CLI)

5.1 Todo List 개발 - CLI 구성

5.2 enum

5.1 Todo List 개발 – CLI(Command Line Interface) 구성

- ✓ 사용자 입력을 받기 위해 TodoConsole 클래스를 정의합니다.
- ✓ TodoConsole 클래스는 TodoCollection 객체를 포함합니다.
- ✓ TodoConsole 클래스에 할일 목록을 표시하는 하는 메소드와 사용자 입력을 받는 메소드를 정의합니다.



5.2 enum type

- ✓ 열거형(enum) 타입은 상수들을 관리하기 위한 객체로 상수의 집합을 정의합니다.
- ✓ 일반 객체는 속성의 변경을 허용하지만 열거형은 속성의 변경을 허용하지 않습니다.
- ✓ 열거형은 속성은 기본적으로 숫자, 문자열만 허용 합니다.
- ✓ 열거형을 이용하면 상수의 수를 제한할 수 있으며 코드의 가독성을 높일 수 있습니다.

```
const korean = 'ko'
const english = 'en'
const japanese = 'ja'
const chinese = 'zh'
const spanish = 'es'

type LanguageCode = 'ko' | 'en' | 'ja' | 'zh' | 'es'

const code: LanguageCode = korean;
```

```
enum LanguageCode {
  korean = 'ko',
  english = 'en',
  japanese = 'ja',
  chinese = 'zh',
  spanish = 'es',
}

const code : LanguageCode = LanguageCode.korean;
```

```
enum ArrowKey {
  Up = 1,
  Down,
  Left = 20,
  Right,
}

console.log(ArrowKey);
```

```
{
  '1': 'Up',
  '2': 'Down',
  '20': 'Left',
  '21': 'Right',
  Up: 1,
  Down: 2,
  Left: 20,
  Right: 21
}
```

```
console.log(ArrowKey.Left);
console.log(ArrowKey[20]);
```

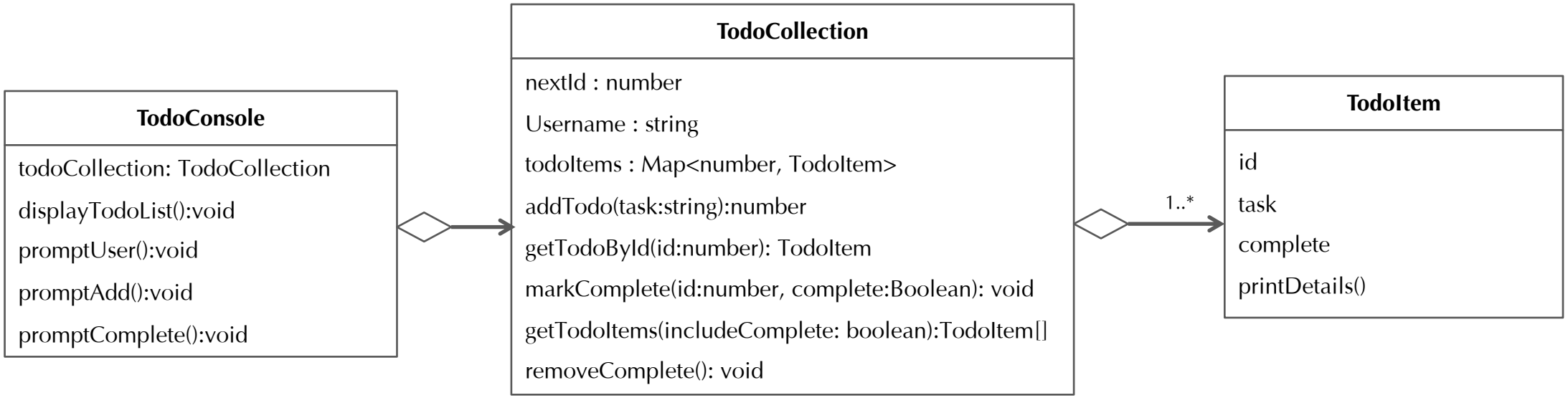
```
20
Left
```

6.Step#06 - Add Functions

6.1 Todo List 개발 - Add, Remove, Complete, Show/Hide

6.1 Todo List 개발 – Add, Remove, Complete, Show/Hide

- ✓ 할 일(Task)을 추가하는 Add 기능을 추가합니다.
- ✓ 완료 된 할일 목록을 제거하는 Remove 기능을 추가합니다.
- ✓ 할 일이 완료 되면 완료 체크 기능을 추가합니다.
- ✓ 완료된 할 일 목록을 보여주거나 보이지 않게 하는 기능을 추가합니다.

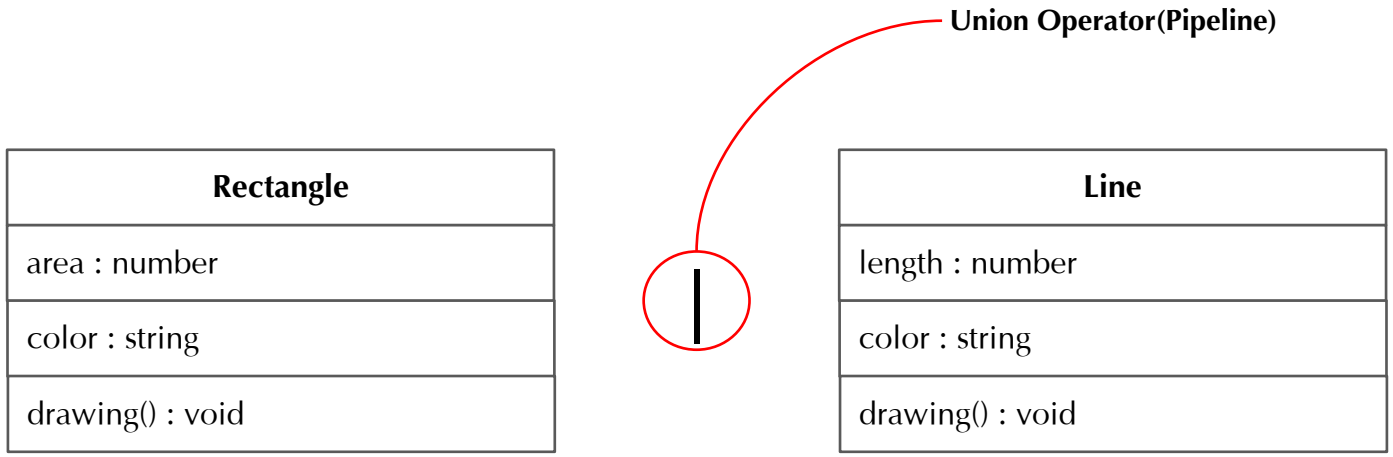


7.Step#07 - Union Type, Type Guard

7.1 Union Type
7.2 Type Guard
7.3 Sorting - Number, String

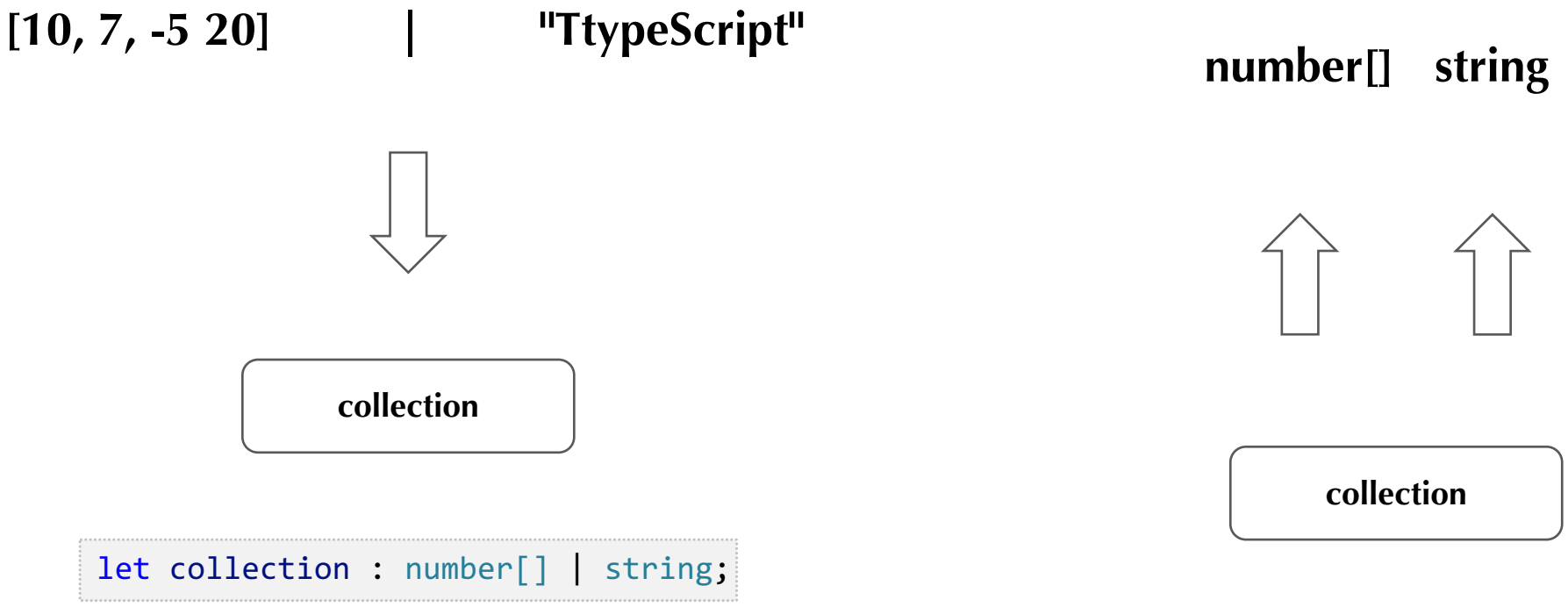
7.1 Union Type

- ✓ TypeScript는 타입들의 조합을 통해 새로운 타입을 정의할 수 있으며 Union Type도 그중 하나입니다.
- ✓ Union Type은 타입 선언에 하나 이상의 타입을 지정하고 해당 타입 중에 하나일 수 있음을 나타냅니다.
- ✓ Union Type의 정의는 | 연산자를 이용해 정의 합니다.
- ✓ Union Type의 멤버 사용은 정의된 모든 타입의 공통적인 멤버들만 사용할 수 있습니다.



7.2 Type Guard

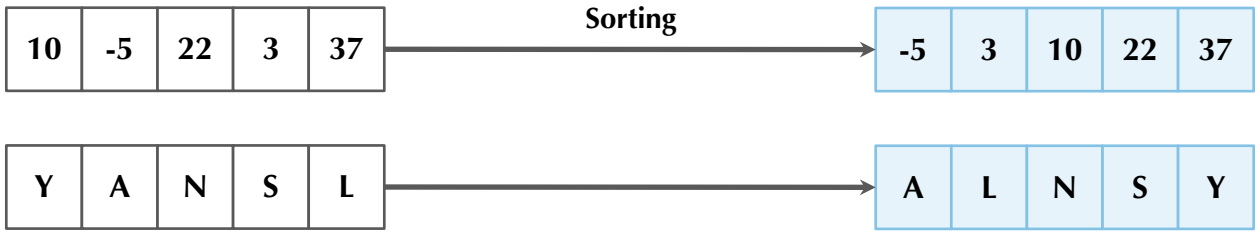
- ✓ Type Guard는 특정 영역(블록) 안에서 해당 변수의 타입을 한정시켜주는 기능입니다.
- ✓ Union Type의 정의는 각 타입이 갖는 고유 멤버는 사용할 수 없습니다.
- ✓ 특정 영역에서 각 타입이 갖는 고유 멤버에 대한 사용은 Type Guard를 이용합니다.
- ✓ Type Guard는 사용자가 정의 하거나 number, string, boolean, Symbol의 경우 typeof 연산자를 이용합니다.

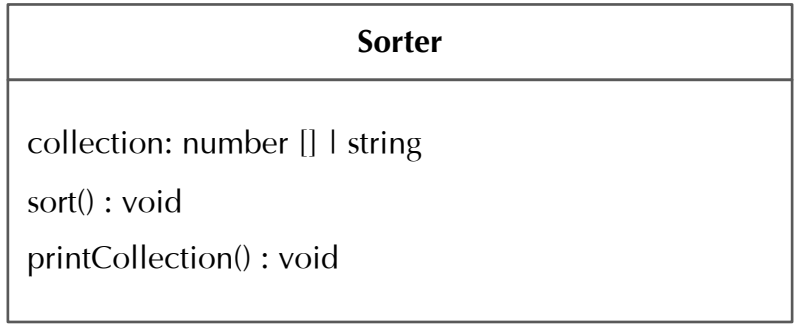



```
let collection : number[] | string;
```

7.3 Sorting – Number, String

✓ 숫자 배열, 문자열의 각 문자에 대한 오름차순 정렬을 구현합니다.
✓ 정렬 방식은 **Bubble sort**를 이용합니다.
✓ 특정 문자열의 문자에 대한 오름차순 정렬은 알파벳 순서입니다.





8.Step#08 - interface

8.1 interface 개요
8.2 interface와 class
8.3 Sorting - Object

8.1 interface 개요(1/2)

✓ 인터페이스는 여러 타입의 속성으로 이루어진 새로운 타입을 정의하는 방법입니다.
✓ 인터페이스도 클래스와 마찬가지로 프로퍼티와 메서드를 갖지만 인터페이스를 이용한 인스턴스는 생성할 수 없습니다.
✓ 인터페이스에 정의하는 메서드는 모두 추상메서드며 abstract 키워드를 사용하지 않습니다.
✓ 인터페이스를 이용하면 변수, 함수, 클래스에 타입을 지정할 수 있습니다.

```
interface TodoItem {
   id : string,
   task : string,
   complete : boolean
}

const todo : TodoItem = {
   id : 'A123’,
   task : 'Study TypeScript!’,
   complete : false
}

interface SumFunc {
   (leftNumber : number, rightNumber): number;
}

const mySum : SumFunc = function( leftNumber : number, rightNumber : number) {
   return leftNumber + rightNumber;
}
```

8.1 interface 개요(2/2) – Optional property, readonly property

✓ 인터페이스의 프로퍼티는 선택적 옵션과 읽기 전용 옵션을 지정할 수 있습니다.
✓ 인터페이스에 정의하는 모든 프로퍼티가 필수 요소가 아닌 경우 ? 를 지정하여 선택적 프로퍼티로 지정합니다.
✓ 인터페이스에 정의하는 특정 프로퍼티에 대해 **readonly**를 지정해 상수처럼 사용할 수 있습니다.

```
interface Shape {
   p1 : number [],
   p2 : number [],
   area? : number;
}

let rectangle : Shape = {
   p1 : [10, 10],
   p2 : [20, 20],
   area : 100
}

let line : Shape = {
   p1 : [10, 10],
   p2 : [20, 20],
}
```

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

let p1 : Point = {
   x : 10,
   y : 20
}

p1.x = 100;
// Cannot assign to 'x' because it is a
// read-only property.
```

8.2 interface와 class

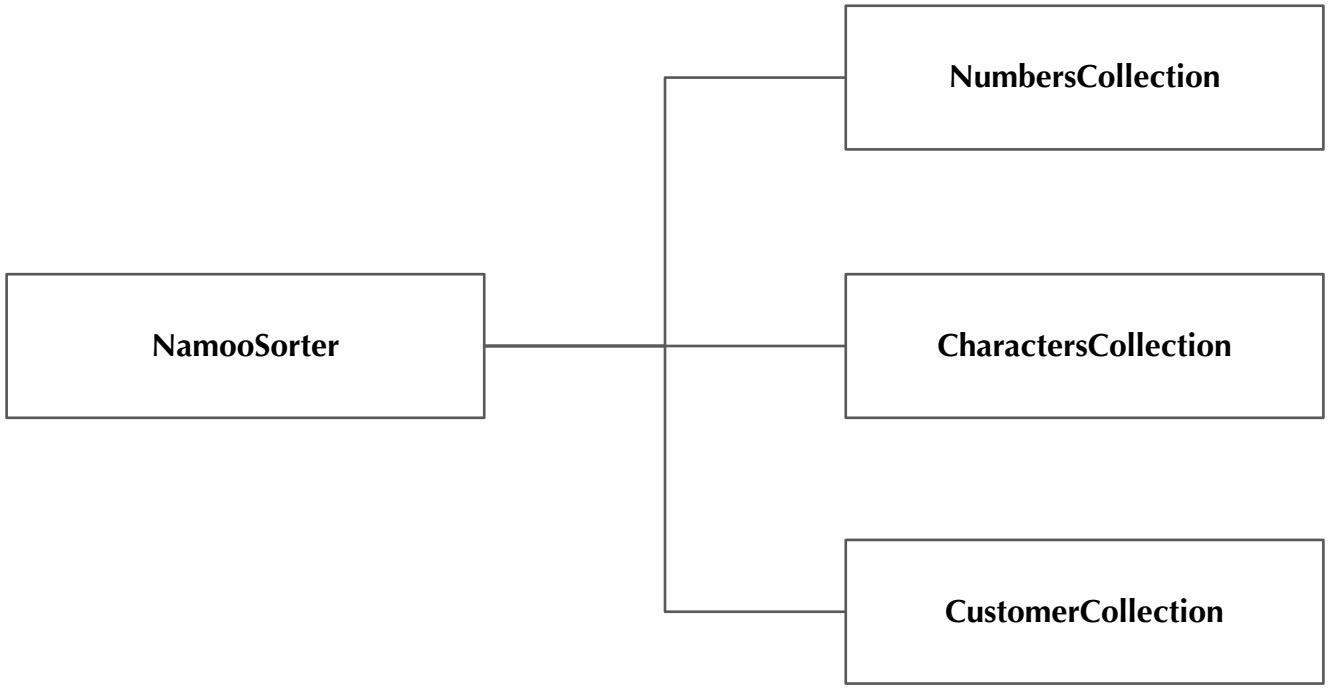
- ✓ 클래스는 implements 키워드를 통해 인터페이스를 구현할 수 있습니다.
- ✓ 인터페이스를 구현한 클래스를 인터페이스에 정의된 추상메소드를 구현해야 합니다.
- ✓ 하나의 클래스는 다수의 인터페이스를 구현(implements) 할 수 있으며 인터페이스간 확장(extends)도 가능합니다.
- ✓ 특정 인터페이스에서 정의한 프로퍼티나 메서드를 갖고 있는 클래스는 해당 인터페이스를 구현한 것으로 인정하며 이를 덕타이핑(duck typing) 이라고 합니다.

```
interface IBhavior {
   play():void;
}
class Soccer implements IBhavior{
   play(){
      console.log('Play Soccer´);
   }
}
class Guitar implements IBhavior {
   play(){
      console.log('Play the guitar´);
   }
}

let bhavior = new Guitar();
bhavior.play(); //Play the guitar
bhavior = new Soccer();
bhavior.play(); //Play Soccer
```

```
interface Drawable {
   drawing():void;
}

interface Sortable {
   sorting():void;
}

class Canvas implements Drawable, Sortable {
   drawing() {
      console.log('Drawing...’);
   };

   sorting() {
      console.log('Sorting...’);
   }
}
```

8.3 Sorting – Object

✓ 기존에 구현한 NamooSorter 프로젝트의 구조를 개선합니다.
✓ 숫자, 문자에 대한 정렬 이외에 특정 객체에 대한 정렬을 구현합니다.
✓ 인터페이스를 통해 확장 가능한 구조를 구성합니다.



End of Document

✓토론

감사합니다...

- 나무소리
- www.namoosori.io