




REACT -

목차

>3/25일	2
1. TypeScript 소개	3
TypeScript의 장점	
Visual Studio Code에서 TypeScript 개발 설정하는 방법	3
1. VS Code 설치	3
2. Node.js 및 TypeScript 설치	3
. TypeScript 코드 컴파일 및 실행	4
3. 기본 문법	6
1. 기본 튜플 선언	9
2. 튜플의 요소 접근	9
3. 튜플의 길이 고정	9
4. Rest 요소를 사용한 추가적인 요소 허용	10
1. 튜플 초기화 후 수정 불가능	10
올바른 점	10
잘못된 점	11
결론:	11
6. 튜플을 함수의 반환값으로 사용	13
7. 튜플을 함수의 인자로 사용	13
8. 다양한 타입의 튜플 사용	13
9. readonly 튜플 사용	14
10. 튜플로 객체처럼 사용하기	14
11. 튜플과 배열의 차이점	14
결론	15
예시로 쉽게 설명:	17
안전하게 사용하려면 타입 검사가 필요:	17
요약:	18
4. 함수 사용법	20
TypeScript 함수 사용법	20
1. 함수 선언 방법	20
1) 기본 함수 선언 (Function Declaration)	20
2) 함수 표현식 (Function Expression)	21
2. 선택적 매개변수 (Optional Parameter)	21
3. 기본값이 있는 매개변수 (Default Parameter)	22
4. 나머지 매개변수 (Rest Parameter)	23
5. 함수의 반환 타입 지정	23

6. 함수 오버로드 (Function Overloading)	24
7. 함수 타입 (Function Type)	25
 정리	25
1. 기본적인 화살표 함수	27
기본 함수 선언	27
화살표 함수	27
2. 매개변수가 없는 경우	28
3. 매개변수가 하나인 경우	28
4. 여러 줄의 함수	28
5. 기본값 매개변수 (Default Parameter)	29
6. 나머지 매개변수 (Rest Parameter)	29
8. 객체 반환 (Implicit Return)	30
 정리	30
5. 인터페이스와 객체 타입 지정	32
TypeScript 인터페이스와 객체 타입 지정	32
1. 객체의 타입을 정의하는 법	32
(1) 타입 별칭 (Type Alias) 사용	32
2. 인터페이스 (interface) 사용법	33
(1) 기본적인 인터페이스 사용	33
(2) 읽기 전용 속성 (readonly)	33
(3) 선택적 속성 (? 사용)	34
3. 인터페이스 확장 (상속)	35
4. 인터페이스와 함수	36
5. 인터페이스와 클래스	36
 정리	37
6. 클래스 기본 개념	38
TypeScript 클래스 기본 개념	38
1. 클래스 정의 및 객체 생성	39
2. 생성자와 속성	40
(1) 생성자 (Constructor)	40
(2) 매개변수를 통한 속성 정의 (단축 문법)	40
3. 접근 제어자 (public, private, protected)	41
(1) public (기본 접근 제어자)	41
(2) private (클래스 내부에서만 접근 가능)	42
(3) protected (상속받은 클래스에서 접근 가능)	43
4. 클래스 상속 (extends)	44
(1) 기본 상속	44
(2) 메서드 오버라이딩 (Method Overriding)	45
(3) super를 사용한 부모 메서드 호출	46
 정리	47
7. 유니온 타입과 타입 가드	48
7. 유니온 타입과 타입 가드 (Union Types and Type Guards)	48
1. 유니온 타입 (Union Types)	48
(1) 기본 사용법	48

(2) 함수에서의 유니온 타입 사용	48
2. 타입 가드 (Type Guards)	49
(1) typeof로 타입 확인	49
(2) instanceof로 타입 확인	50
3. 유니온 타입과 타입 가드 결합	50
📌 요약	51
열거형 (Enums)	52
1. 기본 열거형 사용법	52
2. 사용자 지정 숫자 값	53
3. 문자열 열거형	53
4. 계산된 값	54
5. 열거형 멤버에 접근	54
6. 열거형과 타입의 결합	55
📌 요약	56
8. 모듈과 가져오기 (import / export)	56
8. 모듈과 가져오기 (Import / Export) - 여러 파일에서 코드 사용하기	56
1. 모듈 시스템의 기본 개념	57
(1) 기본 개념	57
2. export 사용법	57
3. import 사용법	58
4. 모듈 경로	60
📌 요약	61
9. 비동기 처리 (async/await)	61
비동기 처리 (Async/Await)	62
1. Promise 개념과 사용법	62
2. Async/Await를 이용한 비동기 코드 작성	63
(1) Async 함수 예시	63
(2) Await 사용 예시	64
(3) 예외 처리 (try/catch)	65
비동기 처리 비교 (Promise vs Async/Await)	66
Promise	66
Async/Await	66
📌 요약	67
10. 실전 예제와 연습문제	67
>4/1일	67
>4/2일	68
>4/3일	68
>4/10일	68
>4/11일	68
>4/12일	68
>4/13일	68
>4/17일	69

> TypeScript 소개

TypeScript란?

JavaScript와의 차이점

VS TypeScript vs JavaScript 차이점

항목	JavaScript	TypeScript	📄
타입 시스템	동적 타이핑(Dynamic Typing)	정적 타이핑(Static Typing)	
실행 방식	바로 실행 가능 (인터프리터)	컴파일 필요 (<code>tsc</code> 로 JS로 변환 후 실행)	
오류 발견 시점	런타임(Runtime)	컴파일 타임(Compile-time)	
개발 도구 지원	제한적	풍부한 자동완성, 리팩토링, 오류 감지 등	
문법 호환성	ECMAScript	ECMAScript + 타입 시스템 (JS 완전 호환)	
학습 곡선	쉬움	다소 높음 (타입 개념 이해 필요)	”
사용 예	소규모 프로젝트, 빠른 프로토타입	대규모 프로젝트, 안정성 요구되는 개발	

TypeScript의 장점

Visual Studio Code에서 TypeScript 개발 설정하는 방법

1. VS Code 설치

Visual Studio Code 공식 사이트에서 최신 버전을 다운로드 후 설치

2. Node.js 및 TypeScript 설치

Node.js 설치 (다운로드)

설치 후 터미널에서 버전 확인

```
node -v
```

```
npm -v
```

TypeScript 전역 설치

```
npm install -g typescript
```

TypeScript 버전 확인

```
tsc -v
```

src 폴더에 app.ts 파일 생성

```
function greet(name: string): string {  
    return `Hello, ${name}!`;  
}  
  
console.log(greet("TypeScript"));
```

. TypeScript 코드 컴파일 및 실행

터미널에서 컴파일 실행

```
PS C:\Users\TJ\Desktop\reactwork> cd ch03
```

```
PS C:\Users\TJ\Desktop\reactwork\ch03> npx tsc app.ts
```

```
PS C:\Users\TJ\Desktop\reactwork\ch03> node app.js
```

TypeScript를 사용하는 이유

TypeScript는 JavaScript의 상위 집합(Superset)으로, 대규모 애플리케이션 개발에 특히 유용합니다. 주요 사용 이유는 다음과 같습니다:

1. 타입 안정성 (Type Safety)

컴파일 단계에서 타입 오류를 잡아줌 → 런타임 에러 감소

예: 숫자에 문자열을 할당하려고 할 때 즉시 오류 표시

```
let age: number = 30;
```

```
age = "30세"; // 컴파일 오류
```

2. 대규모 프로젝트 적합성

코드베이스가 커질수록 유지보수 용이

팀 협업 시 코드 의도가 명확해짐 (함수 인자/반환 타입 명시)

3. 향상된 개발자 경험 (DX)

VS Code 등 IDE에서 자동 완성, 코드 네비게이션, 리팩토링 지원

인텔리센스가 매개변수 타입과 가능한 메서드를 제안

4. 최신 JavaScript 기능 사용 + 추가 기능

ES6+ 기능(화살표 함수, 모듈 등)을 모든 브라우저에서 사용 가능
인터페이스, 제네릭, 데코레이터 등 추가 기능 제공

5. 점진적 도입 가능

기존 JavaScript 프로젝트에 점진적으로 적용 가능

.js 파일을 .ts로 확장자 변경 후 타입 추가 가능

6. 강력한 생태계

React, Angular, Vue 등 주요 프론트엔드 프레임워크 공식 지원

npm 패키지 대부분 타입 정의 파일(@types/) 제공

7. 문서화 역할

타입 정의가 코드의 살아있는 문서 역할 수행

새로운 팀원이 코드베이스 이해 속도 향상

결론: TypeScript는 특히 규모가 크고 장기적으로 유지보수해야 하는 프로젝트에서 코드 품질과 개발 효율성을 크게 향상시킵니다. 초기 학습 곡선이 있지만, 장기적으로 생산성 향상과 버그 감소 효과가 큼니다.

> TypeScript 소개

3. 기본 문법

- 변수 선언 (`let`, `const`)
- 데이터 타입 개념
 - 숫자 (`number`), 문자열 (`string`), 불리언 (`boolean`)
 - 배열 (`Array<T>`, `T[]`)
 - 튜플 (`[number, string]`)
 - 객체 (`{ key: value }`)
 - `Any`, `Unknown`, `Void`

TypeScript 기본 데이터 타입 설명

TypeScript에서는 JavaScript의 동적 타입 지정 방식과 달리 정적 타입을 사용할 수 있습니다. 주요 데이터 타입을 하나씩 설명해드리겠습니다.

1. 숫자 (`number`)

TypeScript의 숫자 타입은 `number` 하나만 존재하며, 정수와 실수를 구분하지 않습니다.

```
let age: number = 25;

let price: number = 99.99;

let hex: number = 0xff;    // 16진수

let binary: number = 0b1010; // 2진수

let octal: number = 0o744;  // 8진수
```

 특징:

`NaN`, `Infinity` 같은 값도 `number` 타입으로 허용됩니다.

```
let notANumber: number = NaN;

let infiniteNumber: number = Infinity;
```

2. 문자열 (string)

문자열 타입은 **string**을 사용하며, 작은따옴표('), 큰따옴표("), 백틱(` ` ` `)을 모두 사용할 수 있습니다.

```
let firstName: string = "John";

let lastName: string = 'Doe';

let fullName: string = `${firstName} ${lastName}`; // 템플릿 문자열
사용 가능
```

3. 불리언 (boolean)

참(true), 거짓(false) 값을 가지는 타입입니다.

```
let isDone: boolean = false;

let hasPermission: boolean = true;
```

JavaScript에서는 `null`, `undefined`, `0`, `""` 등이 `false`로 간주될 수 있지만, TypeScript에서는 `boolean` 타입을 사용해야 명확합니다.

결론적으로, 타입스크립트에서도 `null`, `undefined`, `0`, `""` 등은 `false`로 간주됩니다. 다만, 타입스크립트는 명확한 타입 지정이 필요하고, 조건문에서는 이 값들이 자동으로 `boolean` 타입으로 변환되어 평가된다는 점에서 JavaScript와 차이가 있습니다.

4. 배열 (Array<T>, T[])

TypeScript에서 배열을 정의하는 방법은 두 가지가 있습니다.

T[] 형태 사용

Array<T> 제네릭 사용

```
let numbers: number[] = [1, 2, 3, 4];
```

```
let names: string[] = ["Alice", "Bob", "Charlie"];
```

```
let genericNumbers: Array<number> = [10, 20, 30];
```

 특징:

배열 요소의 타입을 지정하면 다른 타입의 값을 넣을 수 없습니다.

```
let numList: number[] = [1, 2, "hello"]; // ❌ 오류 발생!
```

5. 튜플 (Tuple - [T1, T2, ...])

튜플은 길이와 각 요소의 타입이 고정된 배열입니다.

타입스크립트에서 튜플(Tuple)**은 고정된 크기와 다양한 타입을 가진 배열로 매우 유용하게 사용됩니다. 아래는 많이 사용되는 튜플 문법과 그 활용법을 정리한 것입니다.

차이점 요약

구분	튜플 ([T1, T2])	배열 (T[])
타입	요소마다 다른 타입 가능	모든 요소가 같은 타입
길이	고정	가변
사용 목적	구조화된 데이터 (예: 좌표, 레코드)	같은 타입 목록 (예: 이름 목록)
예시	<code>[string, number]</code>	<code>string[]</code>

1. 기본 튜플 선언

튜플을 선언할 때는 각 요소의 타입을 순서대로 지정합니다.

```
let person: [string, number] = ["Alice", 25]; // 첫 번째는 string, 두 번째는 number
```

- `**person[0]**`은 `string` 타입 (`"Alice"`).
- `**person[1]**`은 `number` 타입 (`25`).

2. 튜플의 요소 접근

튜플은 배열처럼 인덱스로 접근할 수 있습니다.

```
let person: [string, number] = ["Alice", 25];
```

```
console.log(person[0]); // "Alice"
```

```
console.log(person[1]); // 25
```

3. 튜플의 길이 고정

튜플은 길이가 고정되어 있습니다. 선언된 타입의 개수와 정확하게 맞춰야 합니다.

```
let person: [string, number] = ["Alice", 25];
```

```
person = ["Bob"]; // ❌ 오류 발생, 두 번째 요소가 없기 때문
```

```
person = ["hello", 55];는 가능합니다
```

4. Rest 요소를 사용한 추가적인 요소 허용

Rest 요소(...)를 사용하면, 튜플 뒤에 임의의 개수의 추가 요소를 허용할 수 있습니다.

```
let person: [string, number, ...string[]] = ["Alice", 25];  
person.push("Engineer"); // OK: string 추가  
person.push("Developer"); // OK: string 추가  
person.push(30);          // ❌ 오류 발생, number는 허용되지 않음
```

- ...string[]은 문자열만 추가할 수 있도록 제한합니다.
- ...any[]를 사용하면 모든 타입을 추가할 수 있습니다.

```
let person: [string, number, ...any[]] = ["Alice", 25];  
person.push("Engineer"); // OK  
person.push(30);          // OK  
person.push(true);        // OK
```

1. 튜플 초기화 후 수정 불가능

튜플은 불변(**immutable**) 자료형이기 때문에 크기를 변경할 수 없고, 원소의 타입을 변경하는 것도 불가능합니다. 주어진 예시에서 타입을 제대로 지정하고 이를 벗어난 값을 추가하려고 할 때 오류가 발생할 것입니다.

올바른 점

```
let person: [string, number] = ["Alice", 25];
```

정상

```
person[0] = "Bob";
```

잘못된 점

```
person[0] = 30; // 오류: 첫 번째 요소는 string이어야 하므로, number 타입을  
대입할 수 없음
```

```
person[1] = "Bob"; // 오류: 두 번째 요소는 number이어야 하므로, string을  
대입할 수 없음
```

push를 이용해서 새로운 튜플형태로 만들수 있다.

Typescript에서 사용을 권장하지 않은 자바스크립트형태의 배열로 취급됨

```
person.push("Engineer"); // OK: 튜플에 새로운 요소를 추가할 수 있지만 타입이  
맞아야 함
```

```
person.push(30); // OK: 타입에 맞는 값이므로 추가 가능
```

```
console.log(person); // ["Alice", 25, "Engineer", 30]
```

```
person.push("Engineer"); // OK: 'string' 타입 추가
```

```
person.push("Engineer"); // OK: 'string' 타입 추가
```

```
console.log(person); // ["Alice", 25, "Engineer", "Engineer"]
```


결론:

1. 크기는 추가할 수 있지만, 타입에 맞는 값을 추가해야 합니다.
2. 수정은 가능하지만, 기존 튜플의 타입에 맞는 값을 넣어야 하며, 타입이 맞지 않으면 오류가 발생합니다.
3. 튜플의 타입은 초기화 이후 변경할 수 없으며, 튜플의 크기 자체도 변경할 수 없습니다.

6. 튜플을 함수의 반환값으로 사용

함수에서 여러 값을 반환할 때 튜플을 사용할 수 있습니다.

```
function getPersonInfo(): [string, number] {  
    return ["Alice", 25];  
}
```

```
let person = getPersonInfo();  
console.log(person[0]); // "Alice"  
console.log(person[1]); // 25
```

7. 튜플을 함수의 인자로 사용

함수에서 튜플을 인자로 받아 다양한 타입의 값을 한 번에 처리할 수 있습니다.

```
function printCoordinates(coord: [number, number]) {  
    console.log(`X: ${coord[0]}, Y: ${coord[1]}`);  
}
```

```
printCoordinates([10, 20]); // X: 10, Y: 20
```

8. 다양한 타입의 튜플 사용

튜플을 사용하면 다양한 타입을 혼합할 수 있습니다.

```
let data: [string, number, boolean] = ["Alice", 25, true]; //  
string, number, boolean
```

9. **readonly** 튜플 사용

튜플을 **readonly**로 선언하면, 요소를 변경할 수 없습니다.

```
let person: readonly [string, number] = ["Alice", 25];  
  
// person[0] = "Bob"; // ❌ 오류 발생, 수정할 수 없음
```

10. 튜플로 객체처럼 사용하기

튜플을 사용할 때, 인덱스를 객체처럼 사용할 수 있습니다. 단, **as const**로 튜플을 불변으로 만들 수 있습니다.

```
let person = ["Alice", 25] as const;  
  
// person[0] = "Bob"; // ❌ 오류 발생, 튜플은 불변이므로 수정할 수 없음
```

11. 튜플과 배열의 차이점

배열은 모든 요소가 같은 타입을 가질 수 있지만, 튜플은 다양한 타입을 가질 수 있습니다. 그리고 배열은 길이가 가변적인 반면, 튜플은 길이가 고정됩니다.

```
let numbers: number[] = [1, 2, 3]; // 배열, 모든 요소는 number  
  
let person: [string, number] = ["Alice", 25]; // 튜플, 첫 번째는 string, 두 번째는 number
```

결론

튜플은 고정된 개수와 각각 다른 타입의 요소들을 처리할 수 있는 매우 유용한 자료형입니다. 튜플을 통해 함수의 여러 반환 값이나 다양한 타입의 데이터를 다룰 때, 각 요소의 타입을 정확히 지정할 수 있어 타입 안정성을 제공합니다. **Rest** 요소나 **any[]** 등을 활용하여 튜플의 유연성을 높일 수 있습니다.

6. 객체 (Object - { key: value })

객체 타입은 {} 안에 키와 값의 타입을 지정하여 정의할 수 있습니다.

```
let user: { name: string; age: number } = {  
  name: "Alice",  
  age: 30  
};
```

 특징:

특정 키가 존재해야 하며, 타입도 일치해야 합니다.

선택적 속성(?)을 사용하면 특정 키가 없어도 됩니다.

```
let user2: { name: string; age?: number } = { name: "Bob" }; //  
`age` 속성 생략 가능
```


1. any 타입

- ✓ 모든 타입을 할당할 수 있고, 어떤 변수에도 할당할 수 있음.
- ✗ 타입 검사를 건너뛰므로, 잘못된 타입 사용이 발생할 가능성이 큼.

ts

복사편집

```
let value: any;

value = 42;           // OK

value = "Hello";      // OK

value = true;         // OK


let text: string;

text = value; // ✓ 오류 없음 (하지만 위험함!)
```

- any 타입은 타입 검사를 아예 하지 않음.
 - `text = value;`에서 `value`가 실제로 `boolean`이어도 오류가 발생하지 않음.
 - 잘못된 타입이 런타임에서 문제를 일으킬 가능성이 높음.
-

2. unknown 타입

- ✓ 모든 타입을 할당할 수 있음.
- ✓ 다른 타입에 할당하려면 타입 검사를 해야 함 (안전함).
- ✗ 타입 검사를 하지 않고 직접 사용하면 오류 발생.

ts

복사편집

```
let data: unknown;

data = 42;           // OK
```

```
data = "Hello";    // OK
```

```
data = true;       // OK
```

```
let text: string;
```

```
// text = data; // ❌ 오류 발생! unknown은 바로 할당 불가
```

```
// ✅ 타입 검사 후 할당 가능
```

```
if (typeof data === "string") {
```

```
    text = data; // OK
```


```
}
```

- **unknown**은 **any**처럼 모든 값을 저장할 수 있지만, 사용할 때는 반드시 타입 검사를 해야 함.
- **text = data;**는 직접 할당이 불가능하여 오류 발생.
- **if (typeof data === "string")**처럼 타입을 확인한 후에야 사용할 수 있음.

any vs unknown 차이 정리

타입	모든 값 할당 가능	다른 타입으로 할당 가능	타입 검사 필요 여부	안전성
any	✅ 가능	✅ 가능	❌ 검사 없음	❌ 타입 안정성 낮음
unknown	✅ 가능	❌ 바로 할당 불가	✅ 반드시 타입 검사 필요	✅ 타입 안정성 높음

결론

- ****any****는 타입 검사를 건너뛰고 무조건 허용 → 타입 안정성이 낮음.
- ****unknown****은 모든 타입을 받을 수 있지만, 사용할 때 타입 검사를 강제 → 더 안전함.
- **unknown**을 사용하면 실수를 방지할 수 있으므로, **any**보다 권장됨. 

9. Void

`void`는 반환값이 없는 함수에 사용됩니다.

```
function logMessage(message: string): void {  
    console.log(message);  
}
```

 특징:

함수가 값을 반환하지 않을 때 사용.

`void` 타입 변수는 `undefined`만 할당 가능.

```
let unusable: void = undefined;
```

TypeScript의 `void` 타입 상세 설명

`void`는 TypeScript에서 함수가 반환값이 없음을 명시적으로 나타내는 특수한 타입입니다. JavaScript에서 모든 함수는 기본적으로 값을 반환하며, 명시적인 `return`이 없을 경우 `undefined`를 반환합니다. TypeScript는 이를 `void` 타입으로 명확히 표현합니다.

1. 기본 사용법 (함수 반환 타입)

```
function logMessage(message: string): void {  
    console.log(message);  
    // 여기에 return 문이 없음  
}
```

// 사용 예

```
logMessage("Hello, TypeScript!"); // 콘솔에 출력만 하고 아무것도  
반환하지 않음
```

왜 `undefined` 대신 `void`를 사용할까?

의도 표현: 함수가 의도적으로 아무 값도 반환하지 않음을 명시

타입 안정성: 반환값을 사용하려고 하면 컴파일 오류 발생

```
const result: string = logMessage("test"); // 오류: void 타입을  
string에 할당 불가
```

3. 화살표 함수에서의 사용

```
const showError = (errorMsg: string): void => {  
    console.error(errorMsg);  
    // 암묵적으로 undefined 반환  
};
```

4. 콜백 함수에서의 활용

```
function fetchData(callback: (data: string) => void) {  
    // 콜백이 반환값을 사용하지 않음을 보장  
    const data = "받은 데이터";  
    callback(data);  
}
```

```
fetchData((result) => {  
    console.log(result);  
    // 반환값이 없어도 문제 없음  
});
```

TypeScript 함수 사용법

TypeScript에서 함수는 JavaScript와 거의 동일하지만, 타입을 명시적으로 지정할 수 있다는 점이 다릅니다. 이를 통해 더 안전한 코드를 작성할 수 있습니다.

1. 함수 선언 방법

1) 기본 함수 선언 (Function Declaration)

TypeScript에서는 매개변수와 반환값의 타입을 지정할 수 있습니다.

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

 설명:

- `x: number, y: number` → 매개변수 타입
 - `: number` → 반환 타입
 - `return x + y;` → 반환 타입과 일치해야 함
-

2) 함수 표현식 (Function Expression)

함수를 변수에 저장할 수도 있습니다.

```
const multiply = function (x: number, y: number): number {  
    return x * y;  
};
```

또는 화살표 함수로 표현할 수도 있습니다.

```
const divide = (x: number, y: number): number => x / y;
```

2. 선택적 매개변수 (Optional Parameter)

매개변수를 선택적으로 만들려면 `?`를 붙입니다.

```
function greet(name: string, age?: number): string {  
    return age ? `Hello, ${name}. You are ${age} years old.` :  
    `Hello, ${name}.`;  
}
```

```
console.log(greet("Alice")); // "Hello, Alice."
```

```
console.log(greet("Bob", 30)); // "Hello, Bob. You are 30  
years old."
```

 설명:

- `age?: number` → `age` 매개변수는 없어도 됨
- 존재 여부에 따라 다른 메시지를 출력

3. 기본값이 있는 매개변수 (Default Parameter)

매개변수의 기본값을 설정할 수 있습니다.

```
function greet(name: string, greeting: string = "Hello"):  
string {  
    return `${greeting}, ${name}!`;  
}
```

```
console.log(greet("Alice")); // "Hello, Alice!"
```

```
console.log(greet("Bob", "Hi")); // "Hi, Bob!"
```

 설명:

- `greeting: string = "Hello"` → 기본값이 "Hello"
 - `greet("Alice")` 호출 시 "Hello"가 자동 사용됨
-

4. 나머지 매개변수 (Rest Parameter)

여러 개의 값을 하나의 배열로 받을 수 있습니다.

typescript

복사편집

```
function sum(...numbers: number[]): number {  
    return numbers.reduce((acc, cur) => acc + cur, 0);  
}
```

```
console.log(sum(1, 2, 3)); // 6
```

```
console.log(sum(10, 20, 30, 40)); // 100
```

 설명:

- `...numbers: number[]` → 여러 개의 숫자를 배열로 받음
 - `.reduce()`로 배열 요소들을 합산
-

5. 함수의 반환 타입 지정

반환값이 없을 경우 `void`를 사용합니다.

```
function logMessage(message: string): void {  
    console.log(message);  
}
```

 설명:

- `void` → 반환값이 없는 함수

6. 함수 오버로드 (Function Overloading)

같은 함수 이름을 여러 개 선언하여 다양한 매개변수 조합을 지원할 수 있습니다.

```
function getValue(value: string): string;  
function getValue(value: number): number;  
function getValue(value: any): any {  
    return value;  
}
```

```
console.log(getValue("Hello")); // "Hello"  
console.log(getValue(123)); // 123
```

 설명:

- `getValue(value: string): string;` → 문자열을 받으면 문자열 반환
 - `getValue(value: number): number;` → 숫자를 받으면 숫자 반환
 - 실제 구현부에서는 `any`를 사용하여 처리
-

7. 함수 타입 (Function Type)

함수를 변수에 저장할 때 타입을 명확하게 지정할 수도 있습니다.

```
let calculator: (a: number, b: number) => number;
```

```
calculator = (x, y) => x + y;
```

```
console.log(calculator(3, 4)); // 7
```

```
calculator = (x, y) => x * y;
```

```
console.log(calculator(3, 4)); // 12
```

 설명:

- `calculator: (a: number, b: number) => number;` → 함수 타입 정의
 - `calculator`에 다른 함수를 할당 가능
-
-

정리

기능	설명	예제
기본 함수 선언	타입 지정하여 함수 정의	<pre>function add(x: number, y: number): number { return x + y; }</pre>
선택적 매개변수	?를 사용해 선택적으로 입력 가능	<pre>function greet(name: string, age?: number) {}</pre>
기본값 매개변수	매개변수에 기본값 지정	<pre>function greet(name: string, greeting = "Hi") {}</pre>
나머지 매개변수	여러 개의 인자 받기	<pre>function sum(...numbers: number[]) {}</pre>
반환 타입 지정	void 로 반환값 없음을 표시	<pre>function logMessage(): void {}</pre>
함수 오버로드	같은 함수 이름으로 여러 타입 지원	<pre>function getValue(value: string): string;</pre>
함수 타입 정의	함수의 타입을 변수에 저장	<pre>let add: (x: number, y: number) => number;</pre>

TypeScript에서 함수를 더 안전하고 효율적으로 사용하는 방법을 익히셨을 겁니다! 🚀

TypeScript 화살표 함수 (Arrow Function)

화살표 함수는 `=>`를 사용하여 간결하게 함수를 정의하는 방법입니다. JavaScript의 화살표 함수와 동일하지만, TypeScript에서는 매개변수와 반환 타입을 지정할 수 있다는 점이 특징입니다.

1. 기본적인 화살표 함수

일반적인 함수 선언과 비교해보겠습니다.

기본 함수 선언

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

화살표 함수

- `const add = (a: number, b: number): number => a + b;`

📌 차이점:

- `function` 키워드를 사용하지 않고 `=>`를 사용하여 함수를 선언
 - 한 줄 반환이면 `{}` 없이 표현 가능
 - `: number`를 사용해 반환 타입을 명시할 수 있음
-

2. 매개변수가 없는 경우

매개변수가 없는 경우 `()`를 그대로 사용합니다.

- `const sayHello = (): void => {`
 - `console.log("Hello, TypeScript!");`
 - `};`
-

3. 매개변수가 하나인 경우

매개변수가 하나일 때는 `()`를 생략할 수 있습니다.

- `const square = (x: number): number => x * x;`

다음과 같이 `()` 없이도 작성 가능합니다.

- `const square = x => x * x; // 타입스크립트에서 타입 명시는 권장됨`
-

4. 여러 줄의 함수

화살표 함수 내부에서 여러 줄의 코드가 필요하다면 `{}`를 사용하고 `return`을 명시해야 합니다.

- `const multiply = (x: number, y: number): number => {`
 - `const result = x * y;`
 - `return result;`
 - `};`
-

5. 기본값 매개변수 (Default Parameter)

- `const greet = (name: string = "Guest"): string => `Hello, ${name}!`;`
-
- `console.log(greet()); // "Hello, Guest!"`
- `console.log(greet("Alice")); // "Hello, Alice!"`

 설명:

- `name: string = "Guest"` → 기본값을 `Guest`로 설정
 - 인자가 없으면 `Guest`가 자동으로 사용됨
-

6. 나머지 매개변수 (Rest Parameter)

여러 개의 매개변수를 받을 경우 ...을 사용합니다.

- `const sum = (...numbers: number[]): number => {`
- `return numbers.reduce((acc, cur) => acc + cur, 0);`
- `};`
-
- `console.log(sum(1, 2, 3, 4, 5)); // 15`

 설명:

- `...numbers: number[]` → 여러 개의 숫자를 배열로 받음
 - `.reduce()`를 이용해 모든 숫자를 합산
-

8. 객체 반환 (Implicit Return)

객체를 반환할 때는 ()로 감싸야 합니다.

- `const createUser = (name: string, age: number) => ({ name, age`
- `});`
-
- `console.log(createUser("Alice", 25)); // { name: "Alice", age:`
- `25 }`

 설명:

- `{ name, age }`를 ()로 감싸지 않으면 {}가 블록으로 해석됨
- `({ key: value })` 형태로 사용

정리

기능	예제
기본 화살표 함수	<pre>const add = (a: number, b: number): number => a + b;</pre>
매개변수 없음	<pre>const sayHello = (): void => console.log("Hello!");</pre>
매개변수 하나	<pre>const square = x => x * x;</pre>
여러 줄 함수	<pre>const multiply = (x, y) => { return x * y; };</pre>
기본값 매개변수	<pre>const greet = (name = "Guest") => console.log(name);</pre>
나머지 매개변수	<pre>const sum = (...numbers: number[]) => numbers.reduce((a, b) => a + b, 0);</pre>
객체 반환	<pre>const getUser = (name, age) => ({ name, age });</pre>

TypeScript 인터페이스와 객체 타입 지정

****인터페이스(Interface)****는 ****객체가 가져야 할 구조(형태)를 미리 정의 해놓은 것**입니다. 즉, 어떤 객체나 클래스가 특정 인터페이스를 구현(따름)하면, 그 인터페이스에서 정의된 속성과 메서드를 반드시 포함해야 합니다.

1. 객체의 타입을 정의하는 법

객체의 타입을 직접 정의할 수도 있습니다.

(1) 타입 별칭 (Type Alias) 사용

- `type User = {`
- `name: string;`
- `age: number;`
- `};`
-
- `const user: User = { name: "Alice", age: 25 };`

 설명:

- `type` 키워드를 사용하여 객체 타입을 정의
 - `User` 타입을 지정한 변수를 만들 수 있음
-

2. 인터페이스 (interface) 사용법

`interface`를 사용하면 객체의 구조를 정의할 수 있습니다.

(1) 기본적인 인터페이스 사용

- `interface User {`
- `name: string;`
- `age: number;`
- `}`
-
- `const user: User = { name: "Alice", age: 25 };`

 특징:

- `interface`는 객체의 속성과 타입을 미리 정의
- `user` 객체는 `User` 인터페이스를 따름

(2) 읽기 전용 속성 (`readonly`)

속성을 변경하지 못하도록 읽기 전용(`readonly`)으로 설정할 수 있습니다.

typescript

복사편집

- `interface User {`
- `readonly id: number;`
- `name: string;`
- `age: number;`
- `}`
-
- `const user: User = { id: 1, name: "Alice", age: 25 };`
-

- `// user.id = 2; // ❌ 오류 발생 (읽기 전용 속성)`

📌 설명:

- `readonly` 키워드를 사용하면 해당 속성을 변경할 수 없음
- `id` 속성은 한 번 설정되면 수정할 수 없음

(3) 선택적 속성 (? 사용)

속성이 필수가 아니라 선택적으로 가질 수도 있도록 설정하려면 `?`를 사용합니다.

- `interface User {`
- `name: string;`
- `age?: number; // 선택적 속성`
- `}`
-
- `const user1: User = { name: "Alice" }; // ✅ age 없음`
- `const user2: User = { name: "Bob", age: 30 }; // ✅ age 있음`

📌 설명:

- `age?` → `age` 속성은 있어도 되고 없어도 됨
- `user1`과 `user2` 둘 다 유효한 객체

3. 인터페이스 확장 (상속)

인터페이스는 다른 인터페이스를 확장(`extends`)하여 새로운 인터페이스를 만들 수 있습니다.

- `interface Person {`
- `name: string;`
- `age: number;`
- `}`
-
- `interface Employee extends Person {`
- `jobTitle: string;`
- `}`
-
- `const employee: Employee = {`
- `name: "Alice",`
- `age: 30,`
- `jobTitle: "Software Engineer",`
- `};`

 설명:

- `Employee` 인터페이스가 `Person` 인터페이스를 확장
 - `Employee`는 `Person`의 모든 속성을 포함하면서, `jobTitle`을 추가
-

4. 인터페이스와 함수

인터페이스는 함수 타입도 정의할 수 있습니다.

- `interface Greeting {`
- `(name: string): string;`
- `}`
-

- `const sayHello: Greeting = (name) => `Hello, ${name}!`;`
-
- `console.log(sayHello("Alice")); // "Hello, Alice!"`

 설명:

- `Greeting` 인터페이스는 `(name: string) => string` 형태의 함수를 정의
 - `sayHello`가 `Greeting`을 따르므로 매개변수와 반환값이 자동으로 체크됨
-

5. 인터페이스와 클래스

인터페이스는 클래스에 구현될 수도 있습니다.

- `interface Animal {`
- `name: string;`
- `makeSound(): void;`
- `}`
-
- `class Dog implements Animal {`
- `name: string;`
-
- `constructor(name: string) {`
- `this.name = name;`
- `}`
-

- `makeSound(): void {`
- `console.log("Woof! Woof!");`
- `}`
- `}`
-
- `const myDog = new Dog("Buddy");`
- `myDog.makeSound(); // "Woof! Woof!"`

 설명:

- `Dog` 클래스는 `Animal` 인터페이스를 구현(`implements`)
- `makeSound()` 메서드는 반드시 구현해야 함

정리

기능	코드 예제
객체 타입 정의	<code>type User = { name: string; age: number; };</code>
인터페이스 사용	<code>interface User { name: string; age: number; }</code>
읽기 전용 속성	<code>readonly id: number;</code>
선택적 속성	<code>age?: number;</code>
인터페이스 확장	<code>interface Employee extends Person { jobTitle: string; }</code>
함수 타입 인터페이스	<code>interface Greeting { (name: string): string; }</code>

인터페이스와
클래스

```
class Dog implements Animal {}
```

TypeScript의 인터페이스를 활용하면 안정적인 객체 구조를 정의하고 유지보수성을 높일 수 있습니다! 🚀

TypeScript 클래스 기본 개념

6. 클래스 기본 개념

- 클래스 정의 및 객체 생성
- 생성자와 속성
- 접근 제어자 (`public`, `private`, `protected`)
- 클래스 상속 (`extends`)

TypeScript에서는 **객체 지향 프로그래밍(OOP)**을 지원하며, `class` 키워드를 사용하여 클래스를 정의할 수 있습니다.

1. 클래스 정의 및 객체 생성

TypeScript에서 클래스를 정의하고 객체를 생성하는 기본적인 방법입니다.

```
class Person {  
  name: string;  
  age: number;  
  
  constructor(name: string, age: number) {  
    this.name = name;  
    this.age = age;  
  }  
  
  greet() {  
    console.log(`Hello, my name is ${this.name} and I am ${this.age}  
years old.`);  
  }  
}  
  
// 객체 생성  
const person1 = new Person("Alice", 25);
```

```
person1.greet(); // Hello, my name is Alice and I am 25 years old.
```

 설명:

- `class` 키워드를 사용하여 `Person` 클래스를 정의
 - 생성자(`constructor`)를 사용하여 속성 초기화
 - `greet()` 메서드는 인스턴스에서 호출 가능
-

2. 생성자와 속성

(1) 생성자 (Constructor)

생성자는 `constructor()` 메서드로 정의되며, 객체가 생성될 때 자동으로 실행됩니다.

```
class Car {  
  brand: string;  
  year: number;  
  
  constructor(brand: string, year: number) {  
    this.brand = brand;  
    this.year = year;  
  }  
  
  displayInfo() {
```

```
        console.log(`${this.brand} was manufactured in ${this.year}.`);
    }
}
```

```
const car = new Car("Toyota", 2022);
car.displayInfo(); // Toyota was manufactured in 2022.
```

(2) 매개변수를 통한 속성 정의 (단축 문법)

TypeScript에서는 생성자에서 속성을 자동으로 초기화하는 단축 문법을 제공합니다.

```
class Animal {
    constructor(public species: string, private age: number) {}
    getAge() {
        return this.age;
    }
}

const cat = new Animal("Cat", 3);
console.log(cat.species); // ✅ "Cat"
console.log(cat.getAge()); // ✅ 3
//console.log(cat.age); ❌ 오류 (private 속성)
```

 설명:

- `public species: string` → 자동으로 `this.species`에 할당됨
- `private age: number` → 외부에서 접근 불가 (`getAge()` 메서드로 접근 가능)

3. 접근 제어자 (`public`, `private`, `protected`)

TypeScript는 ****접근 제어자(Access Modifiers)****를 제공하여 클래스 멤버(속성 및 메서드)의 접근 범위를 지정할 수 있습니다.

접근 제어자	설명	사용 가능 범위
<code>public</code>	기본 접근 제어자 (생략 가능)	클래스 내부 / 외부 접근 가능
<code>private</code>	클래스 내부에서만 접근 가능	클래스 내부에서만 사용 가능
<code>protected</code>	클래스 내부 및 상속받은 클래스에서 접근 가능	클래스 내부 + 하위 클래스에서 사용 가능

(1) `public` (기본 접근 제어자)

typescript

복사편집

```
class Person {  
  public name: string; // 기본적으로 public (생략 가능)  
  
  constructor(name: string) {  
    this.name = name;  
  }  
  
  public greet() {  
    console.log(`Hello, my name is ${this.name}.`);  
  }  
}  
  
const person = new Person("Alice");  
console.log(person.name); // ✅ "Alice"  
person.greet(); // ✅ "Hello, my name is Alice."
```

(2) `private` (클래스 내부에서만 접근 가능)

typescript

복사편집

```
class BankAccount {
    private balance: number;

    constructor(initialBalance: number) {
        this.balance = initialBalance;
    }

    deposit(amount: number) {
        this.balance += amount;
    }

    getBalance() {
        return this.balance;
    }
}

const account = new BankAccount(1000);
account.deposit(500);
console.log(account.getBalance()); // ✅ 1500
// console.log(account.balance); ❌ 오류 (private 속성)
```

📌 설명:

- `private balance` → 클래스 외부에서 직접 접근 불가
- `getBalance()` 메서드를 통해 간접적으로 접근

(3) **protected** (상속받은 클래스에서 접근 가능)

```
class Animal {
```

```
protected type: string;

constructor(type: string) {
  this.type = type;
}
}

class Dog extends Animal {
  constructor() {
    super("Dog");
  }

  bark() {
    console.log(`The ${this.type} barks.`);
  }
}

const dog = new Dog();
dog.bark(); // ✅ "The Dog barks."
// console.log(dog.type); ❌ 오류 (protected 속성)
```

📌 설명:

- `protected type` → `Animal` 클래스 내부 및 `Dog` 클래스 내부에서만 접근 가능
 - 클래스 외부에서는 `dog.type` 접근 불가능
-

4. 클래스 상속 (**extends**)

TypeScript에서는 `extends` 키워드를 사용하여 클래스를 ****상속(inheritance)****할 수 있습니다.

(1) 기본 상속

```
class Animal {  
    constructor(public name: string) {}  
  
    makeSound() {  
        console.log(`${this.name} makes a sound.`);  
    }  
}
```

```
class Dog extends Animal {  
    constructor(name: string) {  
        super(name);  
    }  
  
    bark() {  
        console.log(`${this.name} barks.`);  
    }  
}
```

```
const dog = new Dog("Buddy");  
dog.makeSound(); // ✅ "Buddy makes a sound."  
dog.bark(); // ✅ "Buddy barks."
```

 설명:

- `Dog` 클래스는 `Animal` 클래스를 상속 (`extends Animal`)
- `super(name)`을 사용하여 부모 클래스의 생성자를 호출

(2) 메서드 오버라이딩 (Method Overriding)

부모 클래스의 메서드를 ****재정의(override)****할 수 있습니다.

```
class Animal {  
  makeSound() {  
    console.log("Some generic animal sound.");  
  }  
}  
  
class Cat extends Animal {  
  makeSound() {  
    console.log("Meow! Meow!");  
  }  
}  
  
const cat = new Cat();  
cat.makeSound(); // ✅ "Meow! Meow!"
```

 설명:

- **Cat** 클래스에서 **makeSound()** 메서드를 재정의하여 동작 변경

(3) **super**를 사용한 부모 메서드 호출

부모 클래스의 메서드를 호출할 수도 있습니다.

```
class Animal {  
  makeSound() {  
    console.log("Some generic animal sound.");  
  }  
}
```

```

}

class Dog extends Animal {
  makeSound() {
    super.makeSound(); // 부모 메서드 호출
    console.log("Woof! Woof!");
  }
}

```

```

const dog = new Dog();
dog.makeSound();
// ✅ "Some generic animal sound."
// ✅ "Woof! Woof!"

```

 설명:

- `super.makeSound();` → 부모 클래스의 `makeSound()` 메서드 실행 후 추가 동작 수행

정리

개념	설명	예제
클래스 정의	<code>class Person {}</code>	<code>new Person()</code>
생성자	<code>constructor(name: string) {}</code>	<code>new Person("Alice")</code>
접근 제어자	<code>public, private, protected</code>	<code>private balance: number;</code>

상속

```
class Dog extends Animal {  
  constructor(name) {  
    super(name);  
  }  
}
```

메서드 오버라이딩

```
super.makeSound();  
makeSound() {  
  super.makeSound();  
}
```

TypeScript의 클래스 기능을 활용하면 객체 지향 프로그래밍 (**OOP**)을 효과적으로 구현할 수 있습니다!

7. 유니온 타입과 타입 가드

- 유니온 타입 (|) 사용법
- `typeof`와 `instanceof`로 타입 확인
-

7. 유니온 타입과 타입 가드 (Union Types and Type Guards)

1. 유니온 타입 (Union Types)

유니온 타입은 여러 타입 중 하나일 수 있음을 나타내는 타입입니다. | 연산자를 사용하여 여러 타입을 결합할 수 있습니다. 이를 통해 하나의 변수나 매개변수가 여러 타입을 가질 수 있도록 할 수 있습니다.

(1) 기본 사용법

- `let value: string | number; // 유니온 타입`
- `value = "Hello"; // 문자열`
- `value = 42; // 숫자`

 설명:

- `value` 변수는 `string` 또는 `number` 타입을 가질 수 있습니다.
- 위 예시에서는 `value`가 문자열과 숫자 모두 가질 수 있음을 명시하고 있습니다.

(2) 함수에서의 유니온 타입 사용

- `function printValue(value: string | number) {`
- `console.log(value);`
- `}`
-
- `printValue("Hello, TypeScript"); // 문자열`
- `printValue(100); // 숫자`

 설명:

- `printValue` 함수는 `string` 또는 `number` 타입의 값을 인수로 받을 수 있습니다.
-

2. 타입 가드 (Type Guards)

타입 가드는 변수가 특정 타입인지 확인하여 해당 타입에 대한 처리를 안전하게 할 수 있는 방법입니다. 이를 통해 유니온 타입을 사용할 때, 런타임에서 타입을 정확하게 추론할 수 있습니다.

(1) `typeof`로 타입 확인

`typeof`는 원시 타입(`string`, `number`, `boolean`, 등)에 대해 타입을 확인할 수 있는 연산자입니다.

- ```
function print(value: string | number) {
```
- ```
  if (typeof value === "string") {
```
- ```
 console.log("String value:", value);
```
- ```
  } else {
```
- ```
 console.log("Number value:", value);
```
- ```
  }
```
- ```
}
```
- 
- ```
print("Hello"); // String value: Hello
```
- ```
print(123); // Number value: 123
```

 설명:

- `typeof`는 변수의 타입이 `string`, `number`, `boolean` 등과 같은 원시 타입인지 확인합니다.
- 유니온 타입 변수에 대해 조건문을 사용하여 타입을 구분할 수 있습니다.
-

## (2) instanceof로 타입 확인

**instanceof**는 객체가 특정 클래스의 인스턴스인지 확인하는 데 사용됩니다.  
주로 객체 타입에 사용됩니다.

```
• class Dog {
• bark() {
• console.log("Woof!");
• }
• }
•
• class Cat {
• meow() {
• console.log("Meow!");
• }
• }
•
• function speak(animal: Dog | Cat) {
• if (animal instanceof Dog) {
• animal.bark(); // Dog 클래스 메서드 호출
• } else {
• animal.meow(); // Cat 클래스 메서드 호출
• }
• }
•
• const dog = new Dog();
• const cat = new Cat();
• speak(dog); // Woof!
• speak(cat); // Meow!
```

 설명:

- **instanceof**는 객체가 특정 클래스의 인스턴스인지를 확인하여 타입 가드로 사용할 수 있습니다.
- 이 방법은 주로 클래스 기반 객체에서 유용합니다.
-

---

### 3. 유니온 타입과 타입 가드 결합

유니온 타입과 타입 가드를 함께 사용하면, 다양한 타입을 처리할 수 있으며, 각 타입에 대해 적절한 로직을 실행할 수 있습니다.

```
• function getLength(value: string | number): number {
• if (typeof value === "string") {
• return value.length;
• } else {
• return value.toString().length; // 숫자도 문자열로 변환
 후 길이 계산
• }
• }
•
• console.log(getLength("Hello")); // 5
• console.log(getLength(12345)); // 5
```

 설명:

- `getLength` 함수는 유니온 타입(`string | number`)을 처리하고, `typeof`를 이용해 각 타입에 맞는 처리 로직을 실행합니다.

---

#### 요약

##### 1. 유니온 타입 (Union Types)

- `|` 연산자를 사용하여 변수나 매개변수가 여러 타입을 가질 수 있게 정의합니다.
- 예: `let value: string | number;`

##### 2. 타입 가드 (Type Guards)

- `typeof`: 원시 타입 (`string`, `number`, `boolean`)에 대해 사용하여 타입을 확인합니다.

- **instanceof**: 객체가 특정 클래스의 인스턴스인지 확인합니다.

### 3. 유니온 타입과 타입 가드 결합

- 유니온 타입을 사용할 때, 타입 가드를 통해 각 타입을 안전하게 처리할 수 있습니다.

타입스크립트에서 유니온 타입과 타입 가드를 잘 활용하면, 타입에 대한 오류를 예방하고 안전한 코드를 작성할 수 있습니다! 🚀

●

## 열거형 (Enums)

열거형(Enums)은 연관된 상수들의 집합을 정의하고, 그 값을 명확하게 구분할 수 있도록 합니다. 타입스크립트에서 열거형은 주로 특정한 값의 집합을 다룰 때 유용하게 사용됩니다.

---

### 1. 기본 열거형 사용법

타입스크립트에서 열거형을 정의하려면 **enum** 키워드를 사용합니다. 기본적으로 숫자 값을 사용한 열거형이 생성됩니다.

```
enum Direction {
 Up, // 0
 Down, // 1
 Left, // 2
 Right // 3
}
```

```
let move: Direction = Direction.Up; // 0
console.log(move); // 0
```

 설명:

- **Direction**이라는 열거형은 네 방향(**Up**, **Down**, **Left**, **Right**)을 정의하고 있으며, 각 항목은 자동으로 0부터 시작하여 숫자 값이 할당됩니다.
- **Direction.Up**은 0이 되고, **Direction.Down**은 1이 됩니다.

## 2. 사용자 지정 숫자 값

열거형의 값은 기본적으로 자동으로 증가하지만, 특정 값으로 지정할 수 있습니다.

```
enum Direction {
 Up = 1,
 Down = 2,
 Left = 3,
 Right = 4
}

let move: Direction = Direction.Left;
console.log(move); // 3
```

 설명:

- **Up, Down, Left, Right**의 값을 직접 지정하여 원하는 숫자 값을 부여할 수 있습니다.

## 3. 문자열 열거형

숫자 대신 문자열을 사용할 수도 있습니다. 이때 각 항목은 문자열 값을 가지며, 명확한 값들을 표현할 수 있습니다.

```
enum Direction {
 Up = "UP",
 Down = "DOWN",
 Left = "LEFT",
 Right = "RIGHT"
}
```

```
let move: Direction = Direction.Up;

console.log(move); // "UP"
```

 설명:

- `Direction.Up`은 이제 문자열 `"UP"`을 가지며, 문자열 값을 통해 열거형을 구분할 수 있습니다.

#### 4. 계산된 값

열거형 항목은 계산된 값을 가질 수 있습니다. 열거형 내에서 다른 항목을 참조하여 값을 계산할 수도 있습니다.

```
enum Direction {

 Up = 1,

 Down = Up * 2, // 계산된 값
}

console.log(Direction.Up); // 1
console.log(Direction.Down); // 2
```

 설명:

- `Direction.Down`은 `Up`의 값을 2배한 계산된 값인 `2`를 갖습니다.

#### 5. 열거형 멤버에 접근

열거형에 정의된 항목에 접근할 때는 `enumName.memberName` 형태로 사용합니다.

```
enum Direction {

 Up = "UP",

 Down = "DOWN",
```

```
 Left = "LEFT",
 Right = "RIGHT"
}
```

```
let move: Direction = Direction.Left;
console.log(Direction[move]); // "LEFT"
```

 설명:

- `Direction[move]`는 `Direction.Left` 값에 해당하는 `"LEFT"` 문자열을 출력합니다.

## 6. 열거형과 타입의 결합

열거형은 다른 타입들과 함께 사용할 수 있습니다. 예를 들어 함수에서 열거형을 파라미터로 사용하면, 열거형 값만 허용되는 제한을 둘 수 있습니다.

```
enum Direction {
 Up = "UP",
 Down = "DOWN",
 Left = "LEFT",
 Right = "RIGHT"
}
```

```
function moveCharacter(direction: Direction) {
 console.log(`Moving character: ${direction}`);
}
```



```
moveCharacter(Direction.Up); // Moving character: UP

moveCharacter("UP"); // 오류: Argument of type '"UP"'
is not assignable to parameter of type 'Direction'.
```

 설명:

- **moveCharacter** 함수는 **Direction** 열거형만을 인수로 받기 때문에, 다른 문자열 값을 넘기면 타입 오류가 발생합니다.

---

 요약

1. 기본 열거형: **enum**을 사용하여 여러 관련된 상수값을 정의할 수 있습니다.
2. 숫자 및 문자열 값: 기본적으로 숫자가 자동으로 할당되며, 문자열 값을 사용할 수도 있습니다.
3. 계산된 값: 열거형 내에서 다른 항목을 참조하여 값을 계산할 수 있습니다.
4. 타입과 결합: 열거형은 함수 인수나 다른 타입과 결합하여 더 구체적인 타입 안전성을 보장합니다.

열거형은 특정 값의 집합을 명확하게 구분하여 사용하기에 매우 유용합니다!

## 8. 모듈과 가져오기 (**import** / **export**)

- 여러 파일에서 코드 사용하기
- 모듈 시스템의 기본 개념

### 8. 모듈과 가져오기 (**Import** / **Export**) - 여러 파일에서 코드 사용하기

타입스크립트에서 모듈을 사용하면 여러 파일에 걸쳐 코드를 분리하여 관리할 수 있습니다. 모듈은 코드의 재사용성을 높이고, 복잡한 애플리케이션을 더 구조적으로 관리할 수 있게 도와줍니다. **import**와 **export**는 모듈 시스템을 활용하는 핵심 기능입니다.

---

#### 1. 모듈 시스템의 기본 개념

타입스크립트는 ES6의 모듈 시스템을 따르며, **export**와 **import** 키워드를 사용하여 모듈 간에 코드를 주고받을 수 있습니다. 모듈은 코드의 네임스페이스를 제공하고, 서로 다른 파일들 간에 데이터를 안전하게 공유할 수 있게 합니다.

##### (1) 기본 개념

- **export**: 다른 파일에서 사용할 수 있도록 변수, 함수, 클래스, 인터페이스 등을 내보냅니다.
- **import**: 다른 파일에서 내보낸 코드(모듈)를 가져옵니다.

---

#### 2. **export** 사용법

## (1) 기본 **export**

변수, 함수, 클래스 등을 내보낼 때 **export**를 사용합니다.

```
// math.ts
export const add = (a: number, b: number): number => {
 return a + b;
};
export const subtract = (a: number, b: number): number => {
 return a - b;
};
```

 설명:

- **add**와 **subtract** 함수는 **math.ts** 파일에서 다른 파일로 내보내졌습니다.

## (2) **export default**

**export default**는 기본적으로 하나의 항목만 내보내야 할 때 사용됩니다. 이 방식은 **import**할 때 이름을 자유롭게 지정할 수 있습니다.

```
// calculator.ts
export default class Calculator {
 add(a: number, b: number): number {
 return a + b;
 }
}
```

 설명:

- `Calculator` 클래스는 `export default`로 내보내지며, 다른 파일에서는 이 클래스를 자유롭게 이름을 지정하여 가져올 수 있습니다.
- 

### 3. `import` 사용법

#### (1) 기본 `import`

내보낸 변수, 함수, 클래스를 다른 파일에서 사용할 때 `import`를 사용합니다.

```
// app.ts
import { add, subtract } from './math';

console.log(add(2, 3)); // 5
console.log(subtract(5, 3)); // 2
```

 설명:

- `add`와 `subtract` 함수는 `math.ts`에서 내보낸 후, `app.ts`에서 가져와 사용됩니다.

#### (2) `import default`

`export default`로 내보낸 항목은 `import`할 때 다른 이름을 사용하여 가져올 수 있습니다.

```
// app.ts
import Calculator from './calculator';

const calc = new Calculator();
console.log(calc.add(10, 20)); // 30
```

 설명:

- `Calculator` 클래스는 `calculator.ts`에서 `default`로 내보내졌기 때문에, `import Calculator`로 가져올 수 있습니다.

### (3) 별칭 (Alias) 사용

모듈을 가져올 때 이름이 겹치거나 긴 경우, 별칭을 사용할 수 있습니다.

```
// app.ts
import { add as sum, subtract as diff } from './math';

console.log(sum(5, 7)); // 12
console.log(diff(7, 3)); // 4
```

 설명:

- `add`와 `subtract` 함수에 각각 `sum`과 `diff`라는 별칭을 사용하여 가져옵니다.
- 

## 4. 모듈 경로

모듈의 경로를 지정할 때, 상대 경로와 절대 경로를 사용할 수 있습니다.

### (1) 상대 경로

```
import { add } from './math'; // 현재 디렉토리에서 math.ts 파일을
가져옴
```

### (2) 절대 경로

절대 경로를 사용하려면 타입스크립트에서 `tsconfig.json` 파일을 통해 `baseUrl`이나 `paths` 옵션을 설정해야 합니다.

```
{
```

```
"compilerOptions": {
 "baseUrl": "./src",
 "paths": {
 "@utils/*": ["utils/*"]
 }
}
```

```
import { add } from '@utils/math'; // 설정된 절대 경로 사용
```

---

## 요약

1. 모듈 시스템: **export**와 **import**를 사용하여 코드의 재사용성과 구조적 관리를 할 수 있습니다.
2. **export**: 변수를 다른 파일로 내보낼 때 사용합니다. **export default**를 사용하면 기본적으로 하나의 항목만 내보냅니다.
3. **import**: 다른 파일에서 내보낸 코드를 가져올 때 사용합니다. **import default**는 기본 내보내기 항목을 가져오며, 이름을 자유롭게 지정할 수 있습니다.
4. 별칭 (**Alias**): **import** 시 변수 이름에 별칭을 붙여 사용할 수 있습니다.
5. 모듈 경로: 상대 경로와 절대 경로를 통해 모듈을 가져올 수 있습니다.

모듈 시스템을 사용하면 코드의 가독성, 재사용성 및 유지보수성을 크게 향상시킬 수 있습니다!

## 9. 비동기 처리 (**async/await**)

- **Promise** 개념과 사용법
- **async/await**를 이용한 비동기 코드 작성

## 비동기 처리 (**Async/Await**)

JavaScript에서 비동기 처리는 웹 애플리케이션을 효율적으로 만드는 데 중요한 역할을 합니다. 특히, **async/await**는 **Promise**와 함께 비동기 코드를 쉽게 작성할 수 있도록 도와주는 문법입니다.

---

### 1. **Promise** 개념과 사용법

**Promise**는 비동기 작업이 완료되었을 때 그 결과 값을 반환하거나 오류를 처리할 수 있도록 하는 객체입니다. 비동기 작업은 네트워크 요청, 파일 읽기, 데이터베이스 쿼리 등 여러 가지 작업에서 발생할 수 있습니다.

**Promise**의 기본 구조는 다음과 같습니다:

```
const promise = new Promise((resolve, reject) => {
 // 비동기 작업
 let success = true; // 작업 성공 여부

 if (success) {
 resolve("작업 성공"); // 성공 시 resolve
 } else {
 reject("작업 실패"); // 실패 시 reject
 }
});
```

```
promise
 .then(result => {
 console.log(result); // "작업 성공"
 })
 .catch(error => {
 console.log(error); // "작업 실패"
 });
```

**Promise**의 상태

- **pending**: 대기 중, 아직 비동기 작업이 끝나지 않음.
- **fulfilled**: 작업 성공, **resolve**가 호출됨.
- **rejected**: 작업 실패, **reject**가 호출됨.

**Promise** 체이닝: 비동기 작업을 여러 개 연결하고 싶을 때 **then**과 **catch**를 사용하여 처리할 수 있습니다.

---

## 2. Async/Await를 이용한 비동기 코드 작성

**async**와 **await**는 **Promise**를 사용한 비동기 처리를 더 간결하고 읽기 쉽게 만들어 줍니다.

- **async**: 함수 앞에 붙여 비동기 함수로 만들 수 있습니다.
- **await**: **async** 함수 내에서만 사용되며, **Promise**가 해결될 때까지 기다립니다.

### (1) Async 함수 예시

```
async function fetchData() {
```



```
 return "데이터가 성공적으로 반환됨!";
}

fetchData().then(result => {
 console.log(result); // "데이터가 성공적으로 반환됨!"
});
```

설명:

- `fetchData` 함수는 `async`로 선언되어 있으며, `Promise`를 반환합니다.
- 이 경우, `return` 값은 자동으로 `Promise.resolve`로 감싸져 반환됩니다.

## (2) `Await` 사용 예시

`await`는 `Promise`가 처리될 때까지 기다립니다. 이 예시는 네트워크 요청을 하는 것처럼 비동기 작업을 기다리는 예입니다.

javascript

복사편집

```
async function fetchData() {
 let response = await fetch("https://api.example.com/data"); //
비동기 작업
 let data = await response.json(); // 응답을 JSON 형태로 변환
 return data;
}

fetchData().then(result => {
 console.log(result); // 서버에서 받은 데이터
}).catch(error => {
 console.log("오류:", error);
});
```

설명:

- `await`는 `fetch` 요청이 완료될 때까지 기다린 후, 응답을 변수 `response`에 저장합니다.
- `await`는 `Promise`가 해결될 때까지 함수 실행을 중단하고 기다림으로 코드가 동기적으로 보이지만 실제로는 비동기 처리가 진행됩니다.

### (3) 예외 처리 (`try/catch`)

`async/await`는 `try/catch` 블록을 사용하여 오류를 처리할 수 있습니다. 이는 비동기 코드에서 발생할 수 있는 예외를 처리하는 데 유용합니다.

```
async function fetchData() {
 try {
 // https://jsonplaceholder.typicode.com/users
 let response = await fetch("https://api.example.com/data");
 if (!response.ok) {
 throw new Error("네트워크 응답이 실패했습니다.");
 }
 let data = await response.json();
 return data;
 } catch (error) {
 console.error("데이터 가져오기 실패:", error);
 }
}
```

```
fetchData().then(result => {
 console.log(result); // 서버에서 받은 데이터
}).catch(error => {
 console.log("오류:", error);
});
```

설명:

- `try` 블록 내에서 `await`를 사용하여 비동기 처리를 기다리고, 오류가 발생하면 `catch` 블록에서 처리합니다.
- 

## 비동기 처리 비교 (Promise vs Async/Await)

### Promise

```
function getData() {
 return new Promise((resolve, reject) => {
 // 비동기 작업
 resolve("성공");
 });
}

getData().then(data => {
 console.log(data);
});
```

### Async/Await

javascript

복사편집

```
async function getData() {
 return "성공"; // Promise.resolve("성공")과 동일
}

getData().then(data => {
 console.log(data);
});
```

```
});
```

차이점:

- **async/await**는 비동기 코드가 더 직관적이고, 읽기 쉬운 구조로 작성됩니다.
  - **Promise**는 **.then()**과 **.catch()**를 사용하여 비동기 처리의 흐름을 처리하지만, 여러 개의 비동기 작업을 다룰 때 코드가 복잡해질 수 있습니다.
- 

## 요약

- **Promise**는 비동기 작업을 처리하는 객체로, 성공과 실패를 처리하는 방법을 제공합니다.
- **async/await**는 **Promise** 기반의 비동기 처리를 더 직관적이고 간결하게 작성할 수 있도록 도와줍니다.
- **async** 함수는 항상 **Promise**를 반환하며, **await**는 **Promise**가 해결될 때까지 기다리는 역할을 합니다.
- 예외 처리는 **try/catch** 블록을 통해 **async/await**를 사용하여 처리할 수 있습니다.

비동기 코드를 작성할 때 **async/await**는 코드 가독성을 높이고, 복잡한 비동기 작업을 다룰 때 유용한 도구입니다!

## 10. 실전 예제와 연습문제

- 간단한 TypeScript 프로젝트 만들기
- JavaScript 코드 변환 연습
-