

TS

“**알아서 잘**
“**딱 끌**” **끔하고**
센 **스있게**
정리하는
TypeScript
핵심 개념

김민석 김민영 김성훈 박성범 양다은 이세영 임현지 전유진 조민경 주다빈

“**알아서 잘**
= **똑같은 꿈하고**
센 스있게
정리하는
TypeScript
핵심 개념”



Contents

Channel. 01 들어가며

- 머리말
- 저자소개
- Notion 링크, PDF 파일 및 QR 코드

Channel. 02 TypeScript 핵심개념

1. TypeScript란?
2. 개발환경 설정방법
3. Type의 종류
4. 배열(Array)
5. 튜플(Tuple)
6. 객체 타입(Object Types)
7. 열거형(Enums)
8. 타입 별칭 & 인터페이스(Type Aliases & Interface)
9. 유니온 타입
10. Type Casting과 Type Assertion
11. 함수(Function)
12. 클래스
13. 제네릭 기초
14. 유틸리티 타입

Channel. 03 마치며

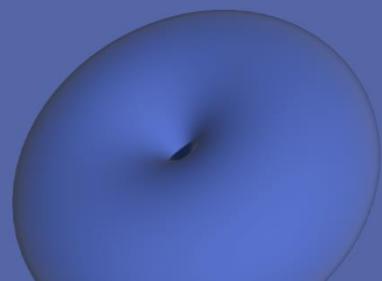
- 참고자료
- 맺음말

01 들어가며

머리말

저자소개

Notion 링크, PDF 파일 및 QR 코드





머리말

?
자바스크립트도 아직 잘 모르는데 타입스크립트를 배우기엔 이른 것이 아닐까?

?
요즘 기업에서 타입스크립트를 할 줄 아는 사람을 뽑는다던데?

?
타입스크립트를 어디서 들어봤는데 여기저기서 자료를 찾긴 귀찮네. 한 번에 쉽게 설명해 준 책이 없을까?

이제 막 자바스크립트를 학습하는 예비 프론트엔드 개발자들에게 타입스크립트는 언젠가 배워야 하지만 아직은 막연한 존재로 느껴집니다. 사실 집필을 하기 전 저희가 그러했습니다. 이 책은 저희와 같이 타입스크립트가 막연하게 느껴질 예비 프론트엔드 개발자들을 위해 쓰였습니다. 타입스크립트가 아직 미지의 언어로 느껴지는 예비 개발자들에게 이 책은 타입스크립트의 첫걸음을 쉽게 내딛게 해줄 디딤돌이 될 것입니다.

타입스크립트는 자바스크립트에 “타입”이란 요소를 추가한 언어입니다. 이 책은 자바스크립트에 추가된 “타입”이라는 요소가 어떤 것인지를 중점적으로 설명하고 있습니다. 따라서 이 책은 자바스크립트에 대한 기초를 어느 정도 이미 학습한 예비 개발자들에게 추천해 드립니다.

참고로 이 책은 이윤을 위한 것이 아닌 오로지 집필하는 저희 예비 개발자들의 성장과 저희와 같이 성장을 하고 싶어 하는 예비 개발자들을 위해 쓰인 무료 책입니다.

그렇다면 타입스크립트란 어떤 언어인지 한 번 알아볼까요?



저자소개

김민석 함께 사용자를 위해 일하는 개발자 🍓
[Github](#)

김민영 확장성을 고려하는 개발자 🦴[Github](#)

김성훈 매일 기록하는 개발자 🎨[Github](#)

박성범 세상을 아름답게 할 개발자 🔍[Github](#)

양다은 모두를 위한 코드를 짜는 개발자 🤖
[Github](#)

이세영 코드를 레코드하는 개발자 🧑[Github](#)

임현지 짜릿하게 배우는 개발자 🐣[Github](#)

전유진 팔로우쉽과 리더쉽을 갖춘 개발자 🐰
[Github](#)

조민경 사람을 좋아하는 개발자 ❤️[Github](#)

주다빈 매일 시도하고 변화하는 개발자 🚀
[Github](#)

👉 저자의 순서는 기여도가 아닌 가나다 순으로 정렬되었습니다 :)



Notion 링크, PDF 파일 및 QR 코드

해당 책은 리디북스, 교보문고 등에서 무료로 다운로드 받으실 수 있습니다. 아래 링크에서 Notion으로 접속하셔서 보실 수도 있으니 참고 바랍니다. (단축 URL의 경우 서비스의 상태에 따라 접속이 안 될 수도 있습니다.)

PDF는 인쇄해서 교재로 사용 가능합니다. 별도의 허락을 구하시지 않으셔도 괜찮습니다.

- 노션링크 : <https://www.notion.so/bumlog/TypeScript-fcf139f54a4b48b3b7634fcc12777a01>
- QR코드 :



- PDF :

알잘딱깔센 TypeScript.pdf 36827.3KB

02 TypeScript 핵심개념

1. TypeScript란?
2. 개발환경 설정방법
3. Type의 종류
4. 배열(Array)
5. 튜플(Tuple)
6. 객체 타입(Object Types)
7. 열거형(Enums)
8. 타입 별칭 & 인터페이스(Type Aliases & Interface)
9. 유니온 타입
10. Type Casting과 Type Assertion
11. 함수(Function)
12. 클래스
13. 제네릭 기초
14. 유ти리티 타입





1. TypeScript란?

 TypeScript(타입스크립트)는 JavaScript에 “정적 타입”이 추가된 언어이다. 따라서 TypeScript는 JavaScript와 타입 두 가지를 모두 포함하고 있는 큰 집합, 즉 JS의 상위 집합(Superset)이라고 할 수 있다.

1.1 TypeScript가 등장하게 된 배경

자바스크립트는 1995년 브랜든 아이크가 10일만에 만든 언어로 유명하다. 그가 처음 자바스크립트를 만들때 자바스크립트는 브라우저에 짧은 코드를 작성하기 위한 용도로 만들어졌다.

그런데 처음 용도와는 달리 JS가 점점 유명해지기 시작하고 브라우저 이외에도 Node.js 등 여러가지 용도로 사용 가능하게 되며 JS로 수 많은 코드를 작성해야 하는 상황이 많이 생기게 된다.

하지만 처음에 짧은 코드를 작성 할 용도로 설계된 JS로 긴 코드를 작성하면 버그가 많이 발생하는 상황이 생겼다. 그 이유는 바로 JS가 “정적 타입 언어”가 아닌 **동적 타입 언어**이기 때문이다. 사실 C나 JAVA와 같이 대부분의 프로그래밍 언어는 **정적 타입 언어**이다.

```
// char이라는 정적 타입으로 선언한 변수 x에는 1바이트 정수 타입의 값만 할당할 수 있다.  
char x;
```

정적 타입 언어는 변수의 타입을 변경할 수 없고 변수에 선언한 타입에 맞는 값만 할당할 수 있다. 그리고 컴파일 시점에 **타입 체크**(선언한 데이터 타입에 맞는 값을 할당했는지 검사)를 진행하여 이 과정을 통과하지 못하면 에러를 발생시키고 프로그램의 실행을 막아준다.

```
// error!(데이터 타입을 숫자로 선언한 변수 num에 다른 데이터 타입을 넣으면 에러가 발생한다.)
var num = 3
num = 'string'
```

하지만 동적 타입 언어는 변수에 어떠한 데이터 타입의 값도 자유롭게 변경할 수 있다. JS에서는 값을 할당하는 시점에 변수의 타입이 동적으로 결정되고 언제든지 자유롭게 변경할 수 있다.

```
// No error
var x = 3
x = 'string'
```

변수에 다른 데이터 타입의 값을 입력하면 개발자의 의도와 상관없이 자바스크립트 엔진에 의해 자동으로 타입이 변경되기도 한다. 물론 이러한 동적 타입 언어는 유연성이 높다는 장점이 있지만 신뢰성이 떨어진다는 단점이 존재한다. 따라서 개발자는 데이터 타입을 오해한 채 버그를 일으키는 코드를 작성하게 될 것이다. 특히 협업을 하는 경우 더욱 그런 문제가 발생할 것이다.

그렇기 때문에 안정적인 프로그램을 만들기 위한 변수 선언을 위해 데이터 타입을 설정하고 체크해주는 타입스크립트의 필요성이 대두되었다.



1.2 TypeScript 학습 방법

앞서서 타입스크립트는 자바스크립트의 상위집합(Superset)라고 하였다. 따라서 자바스크립트 파일 확장자 `.js` 를 `.ts` 로 바꾸어도 사용이 가능하다. 왜냐하면 자바스크립트는 타입스크립트 안에 포함되어 있기 때문이다. 그래서 개발을 입문한지 얼마 되지 않은 사람들은 이런 질문을 할 수도 있다.

| “그렇다면 자바스크립트 대신 타입스크립트를 배우면 되는게 아닌가?”

정답은 간단하다.

| “자바스크립트를 모르면 타입스크립트를 할 수 없다.”

타입스크립트는 자바스크립트와 동일한 문법과 특성을 가지고 있다. 다만 거기에 `Type` 이라는 요소가 추가된 것이다. 시중에는 자바스크립트에 대한 양질의 강의와 교재들이 많이 있다. 따라서 자바스크립트를 먼저 학습한 후, 타입스크립트의 특징을 배운다면 타입스크립트를 훨씬 유용하게 사용할 수 있을 것이다.

또한 브라우저는 타입스크립트 파일을 읽을 수 없다. 그렇기 때문에 타입스크립트로 문서를 작성한 후, 컴파일러를 통하여 타입스크립트 파일을 자바스크립트 파일로 변환을 해주어야 한다. 결국 브라우저가 읽는 것은 자바스크립트 파일인 것이다. CSS를 배운 사람은 Sass를 생각하면 된다.



2. 개발환경 설정방법

2.1 TypeScript 설치방법

 TypeScript를 설치하기 위해서 먼저 Node.js를 설치해 주어야 한다.
Node.js의 기능인 npm이라는 라이브러리를 통하여 TypeScript를 설치해 줄 수 있다.

2.1.1 Node.JS 빠른 설치

아래 Node.js 홈페이지에 들어가신 후, 다운로드를 받는다.

The screenshot shows the official Node.js website. At the top, there's a dark header with the Node.js logo and navigation links: HOME, ABOUT, DOWNLOADS, DOCS, GET INVOLVED, SECURITY, CERTIFICATION, and NEWS. Below the header, a green banner states "Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine." To the right of the banner is a large green geometric graphic. The main content area features the Node.js logo again. Below it, two large green buttons offer "16.15.0 LTS" (Recommended For Most Users) and "18.2.0 Current" (Latest Features). Smaller links for "Other Downloads", "Changelog", and "API Docs" are visible between the buttons. A note at the bottom encourages users to look at the Long Term Support (LTS) schedule.

우측 초록색 버튼(Latest Features)을 클릭하여 가장 최신 버전을 다운로드 받는다.

설치창에서 나오는 모든 항목은 그대로 두고 다음을 눌러 다운로드를 완료하면 Node.js 설치가 완료된다.

2.1.2 npm으로 TypeScript 설치

2.1.2.1 npm이란?

 여기서는 npm을 TypeScript를 사용하는데 필요한 도구 정도로 기억하면 된다.

npm은 세계에서 가장 규모가 큰 라이브러리 / 패키지 관리자 / 설치 프로그램이다.

npm은 무료이며 많은 오픈소스들이 npm을 통하여 공유되고 있다. 따라서 Node.js만 설치하면 따로 다운로드 받거나, 로그인 할 필요없이 바로 사용할 수 있다.



2.1.2.2 설치방법

Visual Studio Code의 터미널 창에 아래 문구를 작성 후 엔터를 누르면, TypeScript가 설치된다.

```
npm install -g typescript
```



The screenshot shows the Visual Studio Code interface with the terminal tab selected. The terminal window displays the following text:

```
문제 출력 터미널 디버그 콘솔 GITLENS
skipLibCheck: true
forceConsistentCasingInFileNames: true

You can learn more at https://aka.ms/tsconfig.json
PS C:\Users\stige\Desktop\Programming\멋쟁이사자처럼(22.04~22.08)\JavaScript_LikeLion> npm install -g typescript []
```

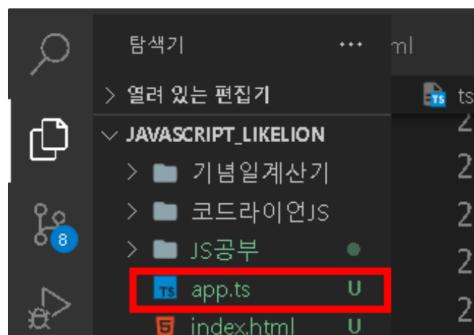
2.2 TypeScript파일 생성하기

- 💡 처음 TypeScript 파일이 생성되는지 테스트를 하며 다양한 문제에 부딪힐 수 있다.
아래의 내용을 통해 문제를 하나하나 해결해 나가보자!

2.2.1 TypeScript파일 생성하기

1. TypeScript 파일생성

Visual Studio Code에서 새 파일을 열어 app.ts 명으로 TypeScript 파일 만듭니다.



2. TypeScript 파일을 JavaScript 파일로 컴파일하기

터미널을 열어준 후, `tsc app.ts` 명령어를 입력하시면 컴파일이 됩니다.

❓ Q. tsc란 어떤 명령어 인가요?

A. tsc는 typescript compiler라는 의미입니다.

즉 타입스크립트(.ts) 파일을 자바스크립트(.js) 파일로 변환(컴파일) 해 줍니다.

❓ Q. 왜 TypeScript 파일을 JavaScript 파일로 컴파일 해줘야 하나요?

A. 브라우저는 TypeScript 파일을 읽을 수 없습니다.

따라서 반드시 브라우저가 읽을 수 있는 JavaScript 파일로 컴파일 해주어야 합니다.

Sass파일을 CSS파일로 컴파일하는것과 비슷한 개념이라고 생각하시면 됩니다.

2. 개발환경 설정방법

The screenshot shows the VS Code interface. On the left is the Explorer sidebar with a tree view of files: 'JAVA SCRIPT_LIKE LION' (expanded), '기념일계산기', '코드라이언JS', 'JS공부' (expanded), 'app.js' (highlighted with a red box), 'app.ts' (version 1.0), 'index.html', and 'tsconfig.json'. The main editor area contains the following code:

```
app.ts > ...
1 function myName(name: string) {
2   console.log(name);
3 }
4
5 myName("Jason");
6
```

Below the editor is the terminal window:

```
문제 ② 출력 터미널 디버그 콘솔 GITLENS
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

새로운 기능 및 개선 사항에 대한 최신 PowerShell을 설치하세요! https://aka.ms/PSWindows
PS C:\Users\stige\Desktop\Programming\멋쟁이사자처럼(22.04~22.08)\JavaScript_LikeLion> tsc app.ts
PS C:\Users\stige\Desktop\Programming\멋쟁이사자처럼(22.04~22.08)\JavaScript_LikeLion>
```

The command `tsc app.ts` is highlighted with a red box in the terminal.

`tsc app.ts` 을 입력하면 app.js 파일이 생성된다.

만약 에러가 발생한다면 아래 내용을 참고.

⚠️ 에러1. about_Execution_Policies

⚠️ 에러2. File 'app.ts' not found

⚠️ 에러3. Error: 중복된 함수 구현

3. 컴파일 된 JavaScript 파일의 실행결과 확인하기

터미널 창에 `node app.js` 를 입력하면 스크립트의 실행된 결과가 터미널창 좌측 하단에 나타난다.

```

1 function myName(name: string) {
2   console.log(name);
3 }
4
5 myName("Jason");
6

```

문제 2 출력 터미널 디버그 콘솔 GITLENS

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

새로운 기능 및 개선 사항에 대 한 최신 PowerShell을 설치하세요! <https://aka.ms/PSWindows>

PS C:\Users\stige\Desktop\Programming\멋쟁이사자처럼(22.04~22.08)\JavaScript_LikeLion> `tsc app.ts`
PS C:\Users\stige\Desktop\Programming\멋쟁이사자처럼(22.04~22.08)\JavaScript_LikeLion> `node app.js`
Jason
PS C:\Users\stige\Desktop\Programming\멋쟁이사자처럼(22.04~22.08)\JavaScript_LikeLion>

만약 에러가 발생한 다면 아래를 참고.

⚠️ 에러4. cannot find module

4. 자동 컴파일 설정하기

앞에서 `tsc` 명령어를 이용하여 TypeScript 파일을 JavaScript파일로 변환하는 방법을 배웠다. 하지만 매번 명령어를 입력하는 것은 매우 귀찮은 일이다. 그래서 이 과정을 자동화 시켜주는 방법이 있다. 바로 `tsc -w` 또는 `tsc --w` 명령어이다. 여기서 w는 watch의 뜻으로, 계속 보고있다는 의미이다.

“`tsc -w 파일경로`” 를 입력하면 해당 파일을 저장할 때마다 자동으로 컴파일을 해준다.

TYPESCRIPT_BOOK

<code>app.js</code>	U
<code>app.ts</code>	U
<code>index.html</code>	U
<code>tsconfig.json</code>	U

문제 출력 터미널 디버그 콘솔 GITLENS

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

새로운 기능 및 개선 사항에 대 한 최신 PowerShell을 설치하세요! <https://aka.ms/PSWindows>

PS C:\Users\stige\Desktop\Programming\멋쟁이사자처럼(22.05~06)\TypeScript_Book> `tsc --w .\app.ts`



에러 1. about_Execution_Policies

에러 발생

Visual Studio Code에서 터미널 창을 열어 `tsc app.ts` 을 입력하여 TypeScript 파일을 JavaScript 파일로 컴파일을 해주려고 한다. 이때 아래와 같은 에러가 발생하였다.

```
PS C:\Users\stige\Desktop\(22.04~22.08)\typeScript> tsc app.ts
tsc : 이 시스템에서 스크립트를 실행할 수 없으므로 C:\Users\stige\AppData\Roaming\npm\tsc.ps
1 파일을 로드할 수 없습니다.
자세한 내용은 about_Execution_Policies(https://go.microsoft.com/fwlink/?LinkId=135170)를
참조하십시오.
오.
위치 줄:1 문자:1
tsc app.ts
+ CategoryInfo : 보안 오류: (:) [], PSNotSupportedException + FullyQualifiedErrorId : Unauth
orizedAccess
```

해결 방법

1. Visual Studio Code 관리자 권한으로 실행합니다.
2. 터미널에 아래 순서대로 입력합니다.

```
> Get-ExecutionPolicy
```

```
> Set-ExecutionPolicy RemoteSigned
```

이때 Restricted가 되어있던 설정이 RemoteSigned으로 변경됩니다.

Restricted는 스크립트 파일을 로드 및 실행할 수 없습니다.

하지만 RemoteSigned은 작성자가 만든 스크립트를 실행 할 수 있습니다. 다만, 인터넷에서 다운로드 받은 스크립트는 신뢰된 배포자에 의해 서명된 것만 실행할 수 있습니다.



에러2. File 'app.ts' not found

에러발생

파일이 발견되지 않는다는 에러가 발생할 수 있다.

```
error TS6053: File 'app.ts' not found.  
The file is in the program because:  
Root file specified for compilation  
Found 1 error.
```

문제해결

혹시 파일이 Directory 안에 들어있지 않은가? 대부분의 경우 잘못된 경로 설정을 할 때 이런 문제가 발생한다. 컴파일을 생성할 때 경로 설정을 정확하게 해주어야 한다.

```
tsc .\TS공부\app.ts
```



Tip. Directory(파일)의 이름 첫글자를 작성 후, Tab을 클릭하면 Directory 전체 이름이 자동완성 된다.



에러3. Error: 중복된 함수 구현

에러발생

정상적으로 TypeScript 파일을 JS파일로 컴파일 해주었다.

하지만 .ts 파일에 아래와 같은 경고 문구가 나타난다.

```
JS공부 > TS test.ts > ...
● 1 ~ function myName(name: string) {
  2   console.log(name);
  3 }
  4
  5 myName("Jack");
  6

문제 1 출력 터미널 디버그 콘솔 GITLENS
▽ TS test.ts JS공부 1
⊗ 중복된 함수 구현입니다. ts(2393) [Ln 1, 열 10]
```

문제 창을 보면 중복된 함수 구현(Duplicate Function Implementation)이라는 경고문구가 발생한 것을 볼 수 있다. 이것은 Visual Studio Code의 기본 설정값으로 인하여 발생하는 문제이다.

문제해결

이것을 없애기 위해서 터미널 창에 `tsc --init` 을 입력한 후 엔터를 누른다.

그리면 아래와 같이 `tsconfig.json` 파일이 생성된다.



이 파일 내부에는 다양한 설정값이 존재하며 추가된 설정값으로 인하여 “중복된 함수 구현” 경고문구는 더 이상 나타나지 않게 된다.



에러4. cannot find module

에러발생

만약 `node app.js` 를 입력하였는데 아래와 같은 경고문구가 나온다면 디렉토리의 경로설정이 잘못되었을 가능성이 높다.

```
Error: Cannot find module 'C:\Users\stige\Desktop\Programming\멋쟁이사자처럼(22.04~22.08)\app.js'
at Module._resolveFilename (node:internal/modules/cjs/loader:939:15)
at Module._load (node:internal/modules/cjs/loader:780:27)
at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:77:12)
at node:internal/main/run_main_module:17:47 {
  code: 'MODULE_NOT_FOUND',
  requireStack: []
}
```

문제해결

디렉토리의 정확한 경로 설정을 해주면 에러는 해결된다.

```
> node .\TS공부\app.js
```



3. Type의 종류

💡 자바스크립트에는 없는 타입스크립트만의 고유한 개념 중 가장 핵심이 되는 것은 타입이다. 타입스크립트와 자바스크립트는 서로 호환되어야 한다. 이를 위해 타입스크립트는 자바스크립트가 사용하는 타입은 물론 타입스크립트의 타입도 사용할 수 있게 제공하고 있다.

3.1 자바스크립트와 타입스크립트의 기본 타입

타입의 종류	JavaScript	TypeScript
수	Number	number
불리언	Boolean	boolean
문자열	String	string
객체	Object	object

타입스크립트는 기본적으로 다음과 같은 타입을 가지고 있다.

| 예제 3-1

```
boolean  
number  
string  
object  
array  
tuple  
enum  
  
any // 모든 타입을 허용. 정해지지 않은 변수 지정 가능.  
  
void // 값이 없음  
  
null  
  
undefined  
  
unknown  
  
never // 도달이 불가능한 코드
```

이 타입들로 자바스크립트의 변수, 함수 따위의 코드에 타입을 정의할 수 있다.

3.2 변수 타입선언

자바스크립트에서는 문자열인 타입의 변수를 숫자 타입으로 변경하는 것이 가능하다. 자바스크립트에서 `let`으로 선언한 변수는 해당 코드에서 그 값이 언제든 변경될 수 있음을 암시한다.

반면, 타입스크립트는 타입 변환이 불가능하다.

자바스크립트는 런타임 도중에 변수 타입이 변경되었을 때 오류가 발생하지 않는데, 타입스크립트는 타입을 강제적으로 선언해 줌으로써 런타임이 아닌 컴파일러 단계에서 오류를 알 수 있다. 이는 생산성의 확실한 향상으로 이어진다.

3.2.1 기본적인 타입 표기

타입스크립트에서는 변수를 선언한 후 콜론 뒤에 타입과 함께 세미콜론을 붙여준다. 이를 **타입주석**(type annotation)이라 한다.

| 예제 3-2

```
let name: string;
let age: number;
```

위처럼 타입주석을 통해 타입을 선언해준 변수는 선언된 타입과 다른 타입의 변수값으로 변경하려 하면 오류가 발생한다.

| 예제 3-3

```
let address: string = '제주';
let age: number = 28;

address = 0; // Error: The type 'number' is not assignable to type 'string'
age = '나이'; // Error!
```

3.2.2 타입추론

타입 표기는 자바스크립트와의 호환성을 위해 생략될 수 있는데, 타입 선언이 없으면 컴파일러가 타입을 추론한다. 아래의 코드를 만나면 타입스크립트 컴파일러가 우측 값에 따라 타입을 지정해 준다. 이를 **타입추론**이라 한다.

예제 3-4

```
let a = true;      // a의 타입을 boolean으로 판단
let b = 'hello'; // b의 타입을 string으로 판단
let c = 1;        // c의 타입을 number로 판단
let d = {};       // d의 타입을 object로 판단

let name: string = "다은"; // 타입을 중복해서 지정
```

예제 3-4의 마지막 코드처럼, 타입스크립트 컴파일러가 타입을 유추할 수 있는 것에 중복하여 명시적으로 타입을 지정하는 것은 피해야 한다.

3.2.3 any

any 타입은 모든 타입의 값을 지정할 수 있다. 숫자, 텍스트, 불린 값 또는 커스텀 타입(사용자가 정의한 타입) 값을 지정할 수 있다. 하지만 타입 체크를 확실히 하기 위해 any를 사용하는 것은 지양된다.

예제 3-5

```
// 모든 값 저장 가능
let a: any = 0;
a = true;
a = 'typescript';
a = {};
```

3.2.4 never 타입

never 타입은 절대 반환을 하지 않는 함수에 사용된다. 도달되지 않는 코드를 나타내며 실행이 종료되지 않고 무한으로 반복되는 함수나 오류를 발생시키기 위해 존재하는 함수가 그 예가 될 수 있다.

예제 3-6

```
const neverTest = () => {
    while(true) {
        console.log("함수가 실행중입니다.");
    }
}
```

위에서 neverTest함수의 타입은 never이다.

never의 또 다른 예시로 다음 함수를 살펴보자.

예제 3-7

```
function sayName(value: string): string {
    if(typeof value === "string") {
        return value;
    } else {
        return value;
    }
}
```

위의 코드에서 else 절의 value에 마우스를 올려 값을 확인해 보면 value : never라고 떠 never 타입임을 확인할 수 있다.

3.2.5 유니온타입

하나의 변수에 지정할 수 있는 타입이 여러 개일 때 **유니온** 타입을 사용한다. 아래와 같이 선언한다.

| 예제 3-8

```
let a: string | number;
```

변수 a는 문자열과 숫자 타입만 허용한다. **any**로 써도 무방하지만 **유니온**으로 필요한 타입만을 지정해 주면 문자열과 숫자 그 외의 값이 들어올 경우 오류를 내보낸다. 예외 처리의 필요성이 사라진다.

💡 두 개 이상의 타입을 가지는 변수는 **any** 타입이 아닌 **유니온** 타입을 사용하는 것이 관습이다. 유니온을 사용하면 타입스크립트 컴파일러가 런타임 도중 잘못된 값이 지정될 경우 이를 감지해 오류를 바로 해결할 수 있다.

3.3 커스텀타입

타입스크립트는 커스텀 타입을 만들 수 있다. type 키워드를 사용하여 새로운 타입을 선언할 수 있고, 타입 별칭(type alias)을 사용하여 이미 존재하는 타입에 다른 이름을 붙여 사용할 수도 있다.

학교에서 학생의 키와 몸무게를 기록하는 페이지를 만든다고 생각하고 키 **Centimeter**와 몸무게 **Kilogram** 타입을 정의해 보자.

| 예제 3-9

```
type Centimeter = number;
type Kilogram = number;
```

위에서 정의한 타입을 가지고 학생을 나타내는 Student 타입을 만들어보자.

| 예제 3-10

```
type Student = {  
    name: string;  
    height: Centimeter;  
    weight: Kilogram;  
}
```

height에는 Centimeter 타입의 별칭을 사용하였고, weight에는 Kilogram 타입의 별칭을 사용해 Student 타입을 선언해 주었다.

다음으로 변수를 만들어 Student 타입을 추가해 주어 초기화한다.

리터럴 표기법으로 인스턴스 student를 만들었다.

| 예제 3-11

```
let student: Student = {  
    name: 'daeun',  
    height: 163,  
    weight: 53  
}
```

만약 student 변수를 초기화할 때 프로퍼티 값이 누락된다면 아래와 같은 메시지가 뜬다.

해당 프로퍼티가 필수가 아닌 선택사항이라면, 타입을 만들 때 프로퍼티 이름 뒤에 물음표를 붙여 해당 프로퍼티가 조건부 프로퍼티임을 선언해 준다.

| 예제 3-12

```
// 프로퍼티 누락
let student: Student = {
  height: 163,
  weight: 53
}
// Property 'name' is missing in type '{ height: number; weight: number; }' but required
in type 'Student'.

type Student = {
  name?: string; // 선택사항
  height: Centimeter;
  weight: Kilogram;
}
// name은 선택사항이 됨. 변수 student는 name 없이 초기화.
let student: Student = {
  height: 163,
  weight: 53
}
```



4. 배열(Array)

💡 배열이란 여러 값을 하나의 변수에 담아 관리하는 자료구조다. 타입스크립트에서는 배열 타입을 배열 타입(array type)과 제네릭 배열 타입(generic array type) 두 가지 형태로 나눈다. 두 가지 형태 중 먼저 배열 타입을 살펴보자.

4.1 배열

배열 타입은 타입스크립트 0.9 버전부터 지원되었으며, 요소 타입(element type)에 대괄호([])를 사용해서 선언한다. 요소 타입으로는 내장 타입인 number, string, boolean 뿐만 아니라 class, interface도 사용할 수 있다. 배열 요소가 모두 number 타입이라면 `number[]` 와 같이 선언해야 한다. 예제를 통해 살펴보자.

예제 4-1

```
let list: number[] = [1, 2, 3, 4, 5];
```

list 변수에 배열 타입으로 number를 추가해 배열을 할당하고 있다.

- `number[]` : 배열 타입
- number : 요소 타입
- `[1, 2, 3, 4, 5]` : 배열 요소, 배열의 길이는 5

예제 4-2

```
let member: string[] = ["김멋사", "이멋사", "박멋사"];
```

배열 요소가 모두 string 타입이므로 배열 타입으로 `string[]` 선언했다. 만약 `Array.prototype.push()` 메서드를 사용해서 배열 요소를 추가하고 싶다면 어떻게 해야 할까? 자바스크립트 문법에 익숙하다면 어렵지 않게 메서드를 사용할 수 있다.

예제 4-3

```
member.push("라이캣");
// result: [ '김멋사', '이멋사', '박멋사', '라이캣' ]
```

또한 배열 요소의 타입이 정해져 있지 않다면 any 타입으로도 지정할 수 있다.

예제 4-4

```
let member: any[] = ["김멋사", 10, true, null];
```

any 타입으로 지정하면 요소 관계없이 배열의 요소로 추가할 수 있지만, 타입이 명확하지 않게 되므로 느슨해지게 된다. any는 피치 못한 상황에 사용하는 경우가 있으나, 제네릭으로도 대응할 수 있으니 웬만하면 any 타입은 피하는 것이 좋다.

예제 4-5

```
let member: (string | number | boolean | null)[] = ["김멋사", 10, true, null];
```

따라서, 예제 4-5와 같이 타입을 제약하기 위해 유니온 타입을 이용해 선언할 것을 권장한다.

4.2 제네릭 배열

제네릭 배열 타입도 배열과 마찬가지로 타입스크립트 0.9 버전부터 지원되었다.

우선, 제네릭 배열 타입의 선언 형태를 간단하게 살펴보자.

| 예제 4-6

```
let str: Array<string> = ["김멋사"];
```

제네릭 배열 타입은 `Array<Type>` 형태로 선언하는데 이때 Type은 뜻 그대로 타입을 의미한다.

| 예제 4-7

```
let str: Array<string | number> = ["김멋사", 10]; // 유니언 타입
let str2: typeof str = ["김멋사", 20]; // 타입 쿼리(type queries)
```

제네릭 배열 역시, 타입을 숫자나 문자열 타입으로 제약하려면 유니온 타입으로 선언한다.

그리고 타입을 참조할 때는 타입 쿼리(type queries)를 이용한다. `typeof` 연산자를 사용하는 데 참조할 변수의 타입을 얻어와 타입을 지정한다.

| 예제 4-8

```
let arr: Array<() => string> = [() => "라이켓", () => "김멋사"];
console.log(arr[0]()); // result: 라이켓
```

제네릭 배열 타입은 객체 타입도 지정할 수 있는데, 예제 4-8처럼 배열 요소를 익명 함수로도 받을 수 있다. `() => string` 형식으로 선언한다.



5. 튜플(Tuple)

사용자의 이름과 나이를 배열에 저장하기 위해 이름은 string으로, 나이는 number로 타입을 지정해서 배열에 저장하는 코드를 튜플로 작성해 보면 아래에 있는 예제 5-1과 같다.

예제 5-1

```
let member: [string, number] = ["김멋사", 10];
console.log(member);
```

배열은 요소의 개수에 제한이 없고, 특정 타입으로 배열 요소의 타입을 강제할 수 있다. 하지만 튜플은 n개의 요소에 대응하는 타입이다. 예를 들어, 예제 5-1에 member 변수에는 `["김멋사", 10]` 두 개의 요소로 이루어진 배열이 저장되어 있으며 첫 번째 요소는 string 타입, 두 번째 요소는 number 타입이다. 튜플 타입 string은 배열의 첫 번째 요소에 대응하고, number는 배열의 두 번째 요소에 대응한다.

즉, `let member: [튜플 타입] = [배열];` 구조로 이루어져 있으며 튜플 타입은 다양한 요소로 이루어진 배열에 대응하는 타입이다.

그리고, 타입스크립트 2.6 이하에서는 튜플 타입에 선언된 개수보다 배열 요소의 개수가 더 많다면 유니온 타입이 적용되었다.

예제 5-2

```
let member: [string, number] = ["김멋사", 10, 100];
```

따라서 예제 5-2 같은 코드가 허용되었기 때문에 튜플 타입은 정확한 요소 개수를 보장하지 않았다. 이러한 동작 방식은 도움이 되지 않는다고 판단되어 타입스크립트 2.7이 되면서 현재와 같이 동작(튜플 타입에 따라 배열의 요소 개수도 고정)하도록 변경되었다. 타입스크립트는 꾸준한 버전 업데이트를 통해 불편한 점을 개선하고 타입 성능을 높이고 있다.



6. 객체 타입(Object Types)

6.1 객체 타입이란?

타입스크립트와 자바스크립트의 차이점 중 하나는 타입스크립트는 객체를 선언할 때 어떤 타입인지 명확하게 정의해야한다는 점이다.

6.1.1 객체 선언

객체는 변수 이름 옆에 : 오른쪽에 {} 또는 new를 사용하여 정의한다. 타입을 지정하는 위치로 객체 생성이 아닌 타입스크립트의 기능을 활용해 작성한다.

예제 6-1

```
const student1: object = { };
const student2 = {} // any 타입
```

타입을 object로 선언하면 typeof 연산자가 기본적으로 모든 타입을 나타낸다. 최상위 타입이기 때문에 그다지 유용하다고 할 수 없다. object는 서술된 값에 관한 정보를 거의 알려주지 않으며, 값 자체가 자바스크립트 객체라고 말해줄 뿐이기 때문에 any 타입보다는 좁은 범위의 타입이다.

타입을 별도로 정해주지 않는다면 타입스크립트는 자동으로 any 타입으로 지정해준다. 명시적으로 `student2: any` 도 가능하다.

예제 6-2

```
let student: {
    name: string,
    grade: number
};

student = {
    name: '학생1',
    grade: 3
};
```

타입을 지정해주면 해당 타입의 객체만 해당 변수에 저장될 수 있다고 알려주는 것이다. 즉 student라는 객체에는 name 필드에 문자열만, grade 필드에는 숫자만 저장될 수 있다.

앞서 정의된 객체인 student의 타입에 맞는 값을 대입하는 방법으로 student에 정의한 객체 타입과 동일한 구조를 가졌기에 위와 같은 student 객체를 저장할 수 있다.

6.2 타입 추론

앞서 객체를 먼저 만들어 필드와 자료형을 지정했다면 표기를 하지 않고도 객체를 만들 수 있다.

예제 6-3

```
const student = {
    name: '김멋사',
    grade: 3
}
```

타입의 표기가 없는 경우 값을 기반으로 타입을 유추해낸다. student 객체의 name은 string이고, grade는 number 자료형으로 추론된다.

6.3 옵션 속성

| 예제 6-4

```
const cat: { type: string, age?: number } = {  
    type: 'Persian'  
};  
cat.age = 2;
```

선택적 속성을 사용하면 해당 필드는 선택적으로 사용할 수 있어 객체 생성 시 지정하지 않고 나중에 값을 추가할 수 있다. 객체를 선언할 때 필드명 뒤에 ?를 붙이면 된다.

6.4 인덱스 시그니처

| 예제 6-5

```
const student: { [index: string]: number } = {};  
  
student.id = 220101;  
student.id = "김멋사"; // Error
```

빈 객체를 만들 때 필드의 자료형을 지정하지 않은 채 인덱스를 사용하면 된다. student라는 객체에 id(string)로 지정하고 해당 값은 220101(number)로 추가할 수 있다.



7. 열거형(Enums)

7.1 열거형이란?

비슷한 종류의 아이템들을 함께 묶어서 표현할 수 있는 수단으로 숫자 또는 문자열 값 집합에 이름을 부여할 수 있는 타입이다. 사용할 수 있는 값을 해당 타입으로 열거하는 기법이다.

| 예제 7-1

```
enum Class {  
    Rock,  
    Scissors,  
    Paper  
}
```

열거형의 이름은 **첫 문자는 대문자로 쓰고 키의 첫 문자도 앞 글자를 대문자로 표시하는 것을 권장한다.** 선언된 열거형은 객체에서 사용하는 것과 동일하게 **점 또는 괄호 표기법**으로 열거형의 값에 접근 할 수 있다. 열거형에는 문자열에서 문자열로 매핑, 문자열에서 숫자로 매핑하는 두 가지 방법이 있다.

7.1.1 숫자 열거형(Numeric enums)

자동으로 열거형의 각 멤버에 적절한 숫자를 추론해 할당하지만, 값을 명시적으로 설정할 수 있다. 상수를 선언만 한다면 숫자 열거형으로 인식해 타입스크립트에서 자동으로 각 멤버에 적절한 숫자를 추론해 할당한다. 예제 7-1처럼 지정된 숫자 값이 없다면 기본적으로 0부터 시작해 1씩 증가한다.

| 예제 7-2

```
enum Class {
  Rock = 0, // 0
  Scissors = 100+1, // 101
  Paper // 102
}
```

숫자 열거형은 `= 숫자`로 초기화하는데 계산한 값도 사용할 수 있다. `Scissors`의 값을 계산한 값으로 초기화했기 때문에 101이 된다. 이때 타입스크립트의 추론에 의해 `Paper`의 값은 작성하지 않아도 `Scissors`의 값에 영향을 받아 값이 102가 된다. 값을 직접 쓰거나 생략하는 것도 가능하지만 되도록 숫자 형이라는 것을 명시해 주는 것이 좋다.

7.1.2 문자열 열거형(String Enum)

| 예제 7-3

```
enum Game{
  Rock = 'ROCK',
  Scissors = 'SCISSORS',
  Paper = 'PAPER'
}
```

문자열 또는 다른 문자열 열거형으로 상수를 초기화해주어야 한다. 숫자 열거형처럼 값이 증가하지 않지만, 값을 읽기 쉽다는 장점이 있다.

7.1.3 이종 열거형

| 예제 7-4

```
enum Game{
    Rock = 'ROCK',
    Scissors = 2,
    Paper
}
```

열거형 값에 문자열과 숫자 값을 혼합하여 사용할 수 있다.

7.2 const enums

| 예제 7-5

```
const enum Game{
    Rock,
    Scissors,
    Paper
}
```

값으로는 문자열 리터럴로만 지정할 수 있으며 숫자 열거형은 안전성을 해치는 문제를 초래한다. 왜냐하면 값이나 키로 열거형에 접근할 때 존재하지 않는 키에도 접근할 수 있기 때문이다. 그렇게 되면 불안정한 결과를 초래하기에 이를 해결하기 위해서는 `const enum`을 사용하는 것이 좋다.



8. 타입 별칭 & 인터페이스(Type Aliases&Interface)

8.1 타입 별칭 (Type Aliases)

8.1.1 타입 별칭이란?

타입에 대한 별칭을 제공하며, 재사용 할 수 있다.

주의해야 할 부분은 타입 별칭은 정의한 타입을 참고할 수 있게 이름을 지어 주는 것이지, 새로운 타입을 생성하는 것이 아니라는 점이다.

8.1.2 타입 별칭 사용

`type 별칭 = 타입;` 으로 정의한다.

| 예제 8-1

```
// 타입만 사용
const food: string = "banana";
const food: string = 1; // Error!

// 타입 별칭을 사용
type Food = string;
const myFood: Food = "banana";
const myFood2: Food = 1; // Error!
```

예제 8-2

```
// 문자열 타입
type Name = string;
const userName: Name = "Jade";

// 문자열 리터럴
type MyName = "Jade";
const userName: MyName = "jade"; // Error!

// 숫자 타입
type MyNumber = number;
const myNumber: MyNumber = 1;

// 유니온 타입
type MyText = string | number;

// 문자열 유니온
type YourText = "hello" | "bye";

// 숫자 리터럴 유니온
type MyNumber = 1 | 2 | 3 | 4 | 5 | 6;

// 객체 리터럴 유니온
type MyObject = { first: 1 } | { second: 2 };

// 함수 유니온
type MyFunction = (() => string) | (() => void);

// 인터페이스 유니온
interface A {a: number;}
interface B {b: number;}
type MyInterface = A | B;

// 튜플 타입
type MyTuple = [string, number];

// 제네릭 타입
type MyGenerics<T> = {
    name: T;
}
```

타입의 자리에는 원시값 이외에도 유니온, 튜플, 함수, 객체 등 인터페이스 레벨의 복잡한 타입이 올 수 있다.

예제 8-3

```
// 객체 타입
type User = {
  name: string,
  age: number
}

// 함수의 파라미터
function getUser (user: User){}
let newUser: User = {
  name: "철수",
  age: 20,
}

getUser(newUser);

// 함수 타입
type AddNumber = (x: number, y: number) => number; // return 해주는 값이 없을땐 void
let addNumber: AddNumber = (x, y) => {
  return x + y;
}
```

타입 별칭은 `string`, `number` 와 같은 원시 타입보다는 객체, 함수 등에 유용하게 사용된다.

8.2. 인터페이스 (Interface)

8.2.1 인터페이스란?

타입 별칭과 비슷하게 새로운 타입을 정의하는 또 다른 방법이다. 객체, 함수, 함수의 파라미터, 클래스 등에 사용할 수 있으며 상호 간에 정의한 약속 혹은 규칙을 의미한다.

`interface` 인터페이스이름 { 속성: 타입; } 으로 정의한다.

8.2.2 인터페이스 속성

8.2.2.1 객체의 스펙

| 예제 8-4

```
// 객체의 스펙(속성과 속성의 타입)
interface User {
    name: string;
    age: number;
}

let user1: User = {
    name: "영희",
    age: 20,
}

let user2 = {} as User;
user1.name = "철수";
user1.age = 20;
```

8.2.2.2 함수 타입

예제 8-5

```
// 함수의 스펙(파라미터, 반환타입 등)
interface AddNumber {
  (x: number, y: number): number; // return 해주는 값이 없을 땐 void
}

let addNumber: AddNumber = (x, y) => {
  return x + y;
}

addNumber(10, 10); // result: 20
```

예제 8-6

```
// 함수의 파라미터
interface User {
  name: string;
  age: number;
}

function getAge(obj: User) {
  console.log(obj.age);
}

let person = {
  name: "철수",
  age: 20,
  gender: "male"
};

getAge(person); // result: 20
```

인터페이스를 인자로 사용할 때, 정의한 프로퍼티와 타입의 조건을 만족한다면, 인자로 받는 객체의 프로퍼티 개수나 순서는 같지 않아도 된다.

8.2.2.3 옵션 속성

인터페이스를 정의할 때 옵션 속성을 사용하면 그 속성을 사용하지 않아도 된다.

속성?: 타입; 과같이 속성 뒤에 ? 를 붙여 사용한다.

| 예제 8-7

```
// 옵션 속성을 사용하지 않은 경우
interface User {
    name: string;
    age: number;
}

function getAge(obj: User) {
    console.log(obj.name);
    console.log(obj.age);
}

let person = {
    name: "철수"
};

getAge(person); // Error!
```

예제 8-8

```
// 옵션 속성을 사용한 경우
interface User {
  name: string;
  age: number;
  gender?: "male" | "female";
}

function getAge(obj: User) {
  console.log(obj.name);
  console.log(obj.age);
}

let person = {
  name: "철수",
  age: 20
};

getAge(person);
```

`getAge` 함수의 인자로 받는 `person`의 프로퍼티에는 `gender` 가 없지만, 인터페이스에 이를 옵션 속성으로 정의했기 때문에 오류가 나지 않는다.

8.2.2.4 readonly 읽기 전용

인터페이스를 사용하여 객체를 생성할 때, 값을 할당한 후 변경할 수 없는 속성을 의미한다.

`readonly` 속성: 타입; 과같이 속성 앞에 `readonly` 를 붙여 사용한다.

예제 8-9

```
// 읽기 전용 속성
interface User {
  readonly name: string;
}

let person: User = {
  name: "철수"
};

person.name = "영희"; // Error!
```

`ReadonlyArray<T>` 타입을 사용하여 읽기 전용 배열을 생성할 수 있다.

예제 8-10

```
// 읽기 전용 배열
let Users: ReadonlyArray<string> = ["철수", "영희"];

Users.splice(0, 1); // Error!
Users.push("현지"); // Error!
Users[0] = "현지"; // Error!
```

8.2.2.5 클래스 타입

`implements`로 미리 정의된 인터페이스를 채택하여 클래스를 정의할 수 있다.

인터페이스로 정의한 메서드나 프로퍼티들은 해당 클래스에 필수적으로 들어가야 한다.

예제 8-11

```
interface User {
    name: string;
    age: number;
    getAge(age: number): void;
}

class newUser implements User {
    name: string = "철수";
    age: number = 20;
    getAge(age: number) {
        console.log(this.age);
    }
    constructor() {}
}

let user1 = new newUser();
user1; // result: {name: "철수", age: 20}
```

| 예제 8-12

```
interface User {
    name: string;
    age: number;
    getAge(age: number): void;
}

class newUser implements User {
    constructor(public name: string, public age: number) {}
    getAge(age: number) {
        console.log(this.age);
    }
}

let user1 = new newUser("영희", 20);
user1; // result: {name: "영희", age: 20}
```

8.3 인터페이스와 타입 별칭의 차이

인터페이스와 타입 별칭의 가장 큰 차이는 타입의 확장 가능, 불가능 여부이다.

8.3.1 선언적 확장

| 예제 8-13

```
interface User {
    name: string;
}

interface User {
    age: number;
} // 같은 이름으로 정의하면 자동으로 확장된다.
```

인터페이스는 새로운 속성을 추가하기 위해 같은 이름으로 재 정의하여 확장할 수 있다. 이를 선언적 확장이라고 한다.

예제 8-14

```
type User = {
  name: string;
}

type User = {
  age: number;
} // Error!
```

타입은 같은 이름으로 재 정의할 수 없다.

8.3.2 인터페이스의 확장

`extends` 를 사용하여 인터페이스 간 확장이 가능하다.

예제 8-15

```
interface User {
  name: string;
}

interface UserInfo extends User {
  age: number;
}

let person: UserInfo = {
  name: "철수",
  age: 20,
};
```

예제 8-16

```

interface User {
  name: string;
}

interface Age {
  age: number;
}

interface UserInfo extends User, Age {
  gender: string;
}

let newUser: UserInfo = {
  name: "영희",
  age: 20,
  gender: "female",
};

```

위 예제처럼 인터페이스를 여러 개 상속받을 수 있다.

예제 8-17

```

// 인터페이스 -> 타입 확장
type User = {
  name: string;
  age: number;
};

interface UserInfo extends User {
  gender: string;
}

let newUser: UserInfo = {
  name: "영희",
  age: 20,
  gender: "female",
};

interface UserInfo2 extends string {} // Error!

```

인터페이스는 타입을 상속받을 수 있지만 리터럴 타입은 불가능하다. 또한 유니온 연산자를 사용한 타입은 `extends`, `implements` 할 수 없다.

8.3.3 타입의 확장

타입 별칭은 `extends` 대신 인터섹션(`&`)을 사용하여 확장할 수 있다. 인터페이스처럼 타입을 상속받아 확장하는 개념이 아닌, 새로운 타입을 정의하는 것이다.

예제 8-18

```
// 타입의 확장
type User = {
    name: string;
};

type UserInfo = User & {
    age: number;
};

let newUser: UserInfo = {
    name: "영희",
    age: 20,
};
```

예제 8-19

```
// 타입 -> 인터페이스 확장
interface User {
    name: string;
    age: number;
}

type UserInfo = User & {
    gender: string;
};

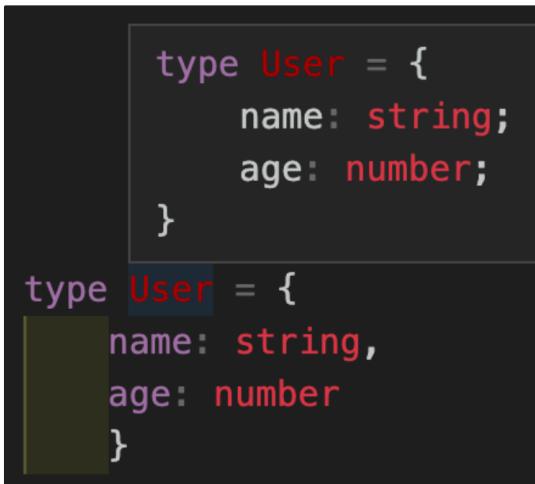
let newUser: UserInfo = {
    name: "영희",
    age: 20,
    gender: "female",
};
```

타입뿐 아니라 인터페이스도 확장이 가능하다.

8.3.4 VScode preview 속성 정보

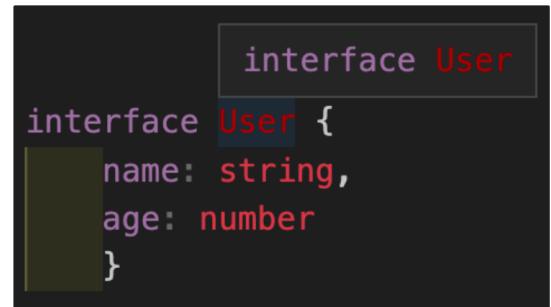
```
// 타입 선언  
type User = {  
    name: string,  
    age: number  
}
```

```
// 인터페이스 선언  
interface User {  
    name: string,  
    age: number  
}
```



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

해당 타입의 전체 속성 정보 조회 가능



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

인터페이스 명만 조회 가능

VScode의 프리뷰 기능을 통해 해당 타입의 속성 정보를 조회할 수 있다. 인터페이스는 명을, 타입 별칭은 객체 리터럴 타입을 보여준다.

8.3.5 인터페이스 vs 타입 별칭

TypeScript의 공식 문서에서는 가능하다면 인터페이스를 사용하고, 인터페이스로 표현할 수 없거나 유니온, 튜플을 사용해야 하는 상황이라면 타입 별칭을 사용하도록 권장하고 있다.



9. 유니온 타입

 Union Type은 TypeScript가 가지는 타입 중 하나로 하나의 값이 여러개의 타입을 가지는 경우 사용한다. OR 연산자처럼 'A이거나 B이다' 라는 의미를 갖는다.

9.1 유니온타입 지정

파이프 문자를 추가해 여러 개의 타입을 지정해 줄 수 있다. OR 연산자와 같이 `A | B`로 표현이 된다.

| 예제 9-1

```
let text: string | number = 22;
text = '22';
```

9.2 인터페이스 유니온

유니온 타입이 인터페이스를 연결했을 때 모든 타입의 공통 속성에만 접근할 수 있다.

| 예제 9-2

```
interface Ujin {
  name: string;
  age: number;
}

interface Dabin {
  name: string;
  character: string;
}
```

```
function combine(person: Ujin | Dabin) {
    person.name; // 정상 동작
    person.age; // 타입오류
    person.character; // 타입오류
}
```

위 코드를 보면 `combine()` 함수 매개변수의 타입을 `Ujin`, `Dabin` 인터페이스의 유니온 타입으로 지정하였다. 매개변수의 타입이 `Ujin` 도 될 수 있고 `Dabin` 도 될 수 있으니 객체의 속성인 `name`, `age`, `character` 모두 사용할 수 있을 것이라 생각하지만, `combine()` 함수를 호출할 때 어떤 타입이 올지 알 수 없기 때문에 어떤 타입이 들어오든 오류가 안 나는 방향으로 타입을 추론하게 된다. 모든 타입의 공통적인 속성에만 접근할 수 있기 때문에 `person.name`은 정상 작동하지만, `person.age` 와 `person.character`는 타입 오류가 난다. 이때 필요한 것이 유니온 타입 가드이다.

9.3 유니온 타입 가드

유니온 타입으로 선언한 변수나 객체를 사용해야 할 경우 그대로 사용하면 오류가 날 수 있다. 타입스크립트는 유니온 타입을 이해할 뿐 유니온 타입 내에 무엇이 있는지는 분석하지 못한다. 이런 경우 런타임 타입 검사를 추가하여 타입이 어느 쪽에 해당하는지 판정을 해줘야한다. 이러한 과정을 타입 가드라고 한다.

9.3.1 원시 타입 식별

원시 타입은 `typeof` 연산자를 이용해 타입 검사를 할 수 있다.

| 예제 9-3

```
function combine(input1: number | string, input2: number | string) {
    let result;
    if (typeof input1 === 'number' && typeof input2 === "number") {
        result = input1 + input2;
    } else {
        result = input1.toString() + input2.toString();
    }
    return result;
}

console.log(combine('hello', 'world')) // helloworld
console.log(combine(10, 10)) // 20
```

9.3.2 클래스 객체 식별

생성자 함수를 반환하는 class 객체는 `typeof`로 검사하면 'object'만을 반환하기 때문에 `instanceof` 연산자를 이용해 타입 검사를 한다. `instanceof` 연산자는 생성자의 프로토타입이 객체의 프로토타입 체인에 있는지 검사한다.

예제 9-4

```
class Ujin { }
class Dabin { }

const combine(user: Ujin | Dabin) {
  if (user instanceof Ujin) {
    user.jjeu_jjeu() // user가 Ujin 클래스의 객체
  } else {
    user.bong_gug() // user가 Dabin 클래스의 객체
  }
}
```

9.3.3 일반 객체 식별

자바스크립트의 객체리터럴과 달리 타입스크립트에서는 객체 타입을 지정할 때 인터페이스를 사용하여 객체의 모양을 지정할 수 있다.

예제 9-5

```
interface Cat {
  meow(): string
}

interface Dog {
  bowwow(): string
}
```

```
function checkType(pet: Cat | Dog) {  
    if ("meow" in pet) {  
        (pet as Cat).meow()  
    } else {  
        (pet as Dog).bowwow()  
    }  
};
```

if 문의 조건을 통해서 pet 타입을 체크였지만 if 문 내부에서 as 문을 통해 한 번 더 어떤 객체인지 알려주어야 한다.

9.3.4 null 채크

문자열의 값이 존재하지 않는 경우에 사용한다. string | null이라고 유니온 타입을 지정했을 때 값이 null이어서 string으로 사용할 수 없거나 null이 아닌 경우 예외 처리를 하고 싶을 때 사용한다.

| 예제 9-6

```
function nullCheck(val: string | null): number {  
    if (val != null) {  
        return val;  
    } else {  
        return 0;  
    }  
}
```



10. Type Casting과 Type Assertion

10.1 타입 캐스팅(Type Casting)

JAVA나 C++등 다른 프로그래밍언어에서 타입 캐스팅이란 형 변환을 의미한다. 타입스크립트에서도 특정 타입임을 단언해야하는 상황이 있는데 특별한 검사나 데이터 재구성을 하지 않기 때문에 Type Casting과는 별개의 개념이다.

10.2 타입 어설션(Type Assertion)

Type Assertion이란 시스템이 추론한 타입의 내용을 변경하는 것이다. 실행시간에 어떤 동작이 일어날것인지를 내포하는 Type Casting보다 Type assertion이 더 적합한 표현이기 때문에 헷갈리지 않아야 한다. 타입 어설션은 `<>` 표기법과 `as` 연산자를 이용한 두 가지 방법이 있다.

10.2.1 `<>` 표기법

`<type>` 을 변수 앞에 작성한다.

| 예제 10-1

```
var val:any;  
var foo1 = <string>val
```

10.2.2 `as` 연산자

`as type` 을 변수 뒤에 작성한다.

| 예제 10-2

```
var val:any;  
var foo2 = val as number
```

<> 표기법은 JSX 문법에서 헷갈리기 때문에 `as` 연산자 사용을 권장한다.

10.2.3 주의해야할 점

| 예제 10-3

```
const str = 'hello world';  
console.log((str as number[]).push(123)); //Error
```

문자열로 선언되어있는 `str` 변수를 `as number[]` 를 통해 타입 에러를 무시하고 숫자형을 넣었을 때 런타입 과정에서 에러가 발생한다. 코드 타입이 확실해야 오류가 발생하지 않으므로 주의해야한다.



11. 함수(Function)

💡 이번 장에서는 함수에서 JavaScript와 TypeScript 문법을 비교해보며, 타입을 어떻게 설정하는지 알아보도록 하자.

11.1 함수의 인자

JavaScript에서는 함수에 매개변수를 적고 인자를 넘겨주지 않으면 `undefined`로 값을 리턴해주지만, TypeScript에서는 함수에 매개변수를 적고 인자를 넘겨주지 않으면 `Error`가 발생한다. 이는 TypeScript에서 매개변수를 선언하면 '**필수적으로 인자를 입력해야 한다(필수값)**'는 것을 의미한다.

💡 **필수값이란?** 함수의 매개변수를 선언하면 `null` 혹은 `undefined`라도 인자로 넣어야 한다는 뜻이다. 컴파일러는 정의한 매개변수 값이 넘어왔는지 확인한다. 더불어 정의된 매개변수 값만 받을 수 있고 추가로 인자를 받을 수 없다.

11.2 함수의 기본 타입 선언

JavaScript의 문법으로 함수를 작성할 때는 예제 11-1처럼 작성한다.

| 예제 11-1

```
function sum(number1, number2){  
    return number1 + number2;  
    console.log(number1+number2);  
}  
  
console.log(sum(10,20)); //result: 30  
console.log(sum(10,20,30)); //result: 30  
console.log(sum(10)); //result: NaN
```

TypeScript는 JavaScript와는 다르게 각 매개변수 뒤에 `:타입` 을 입력해주고 꽂호 뒤에는 함수가 반환하는 `:타입` 을 입력한다. 반환하는 타입이 없는 경우 `:void` 로 입력한다.

타입 종류는 `Boolean`(불리언), `Number`(숫자), `String`(문자열), `Array`(배열), `유니언 타입`(다중타입: 문자열과 숫자를 동시에 가지는 배열), `any`(타입을 단언할 수 없을 때)가 있다.

TypeScript에서는 어떤 방식으로 작성되는지 아래의 예제를 통해 알아보자.

11.2.1 반환 값이 있는 경우

함수의 반환 값이 있는 경우 반환하는 자료형의 타입을 기재한다.

예제 11-2

```
function sum(number1: number, number2: number): number{
    return number1 + number2;
}
console.log(sum(10,20)); //result: 30
console.log(sum(10,20,30)); //Error!(파라미터 개수가 많아서 에러)
console.log(sum(10)); //Error!(파라미터 개수가 너무 적어서 에러)
```

예제 11-3

```
function isChildren(age: number): boolean{
    return age < 19;
}
console.log(isChildren(47)); //result: false
console.log(isChildren(5)); //result: true
```

11.2.2 반환 값이 없는 경우 `void`

반환 값이 없는 경우 `void`를 기재한다.

| 예제 11-4

```
function sum(number1:number, number2:number): void{
    console.log(number1 + number2);
}
```

11.3 선택적 매개변수 ‘?’

JavaScript에서는 예제 11-5와 같이 매개변수의 개수만큼 인자를 넘기지 않아도 된다. 하지만 함수의 기본 타입을 선언하게 되면 JavaScript의 특성과는 다르게 Error가 발생한다. 예제 11-5는 JavaScript의 문법으로 작성한 예시이다.

| 예제 11-5

```
function morning(name){
    return `Good morning ${name || 'everyone'} `;
}

console.log(morning()); //result: Good morning everyone
console.log(morning('mjo')); //result: Good morning mjo
console.log(morning(123)); //result: Good morning 123
```

결과를 확인하면 JavaScript로 작성한 코드는 별도의 Error가 발생하지 않는다.

TypeScript에서 위와 같은 JavaScript의 특성을 살리기 위해 **선택적 매개변수**가 등장했다. 변수명 앞에 '?'를 붙이면 그 매개변수는 Optional한 값이 되고 Optional한 값은 인자를 넘기지 않아도 에러가 발생하지 않는다.

예제 11-6

```
function morning(name?: string): string {
    return `Good morning ${name || 'everyone'}`;
}

console.log(morning()); //result: Good morning everyone
console.log(morning('mjo')); //result: Good morning mjo
console.log(morning(123)); //Error!(타입에러)
```

TypeScript의 매개변수 타입 확인 절차는 거치면서 JavaScript의 특성은 살리는 모습을 확인할 수 있다.

여기서 선택적 매개변수를 사용할 때 주의할 점이 있는데, **선택적 매개변수가 필수 매개변수보다 앞에 위치하면 에러가 발생**한다는 사실이다. 만약 에러 없이 선택적 매개변수를 앞에 위치하고 싶다면 '?를 제거하고 '| undefined'를 선언하면 된다. 예제 11-7이 그 예시이다.

예제 11-7

```
function morning(name: string | undefined, time: number): string {
    return `Good morning ${name || 'everyone'}. Time is ${time || 8}.`;
}

console.log(morning()); //Error!
console.log(morning('mjo', 7)); //result: Good morning mjo. Time is 8.
console.log(morning(undefined, 123)); //result: Good morning everyone. Time is 123.
```

11.4 매개변수 초기화

타입을 적어주는 것과 다른 방법으로 **매개 변수를 선언할 때 값을 미리 초기화** 할 수 있다. 필수 매개 변수 앞에 위치해도 정상 작동을 하며 따로 타입을 지정해주지 않아도 된다.

| 예제 11-8

```
function sum(a: number, b = 2022): number {
    return a+b;
}
console.log(sum(10,undefined)); //result: 2032
console.log(sum(10,20,30)); //Error!(파라미터 개수가 많아서 에러)
console.log(sum(10)); //result: 2032
```

11.5 REST 문법이 적용된 매개변수

전개문법으로 받은 매개변수는 **배열**이기 때문에 타입을 배열로 받는다.

| 예제 11-9

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

console.log(sum(10, 20, 30)); //result: 60
console.log(sum(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)); //result: 55
```

11.6 this

this의 타입을 정할 때는 함수의 첫 번째 매개변수 자리에 `this`를 쓰고 타입을 입력한다.

예제 11-10

```
function 함수명(this: 타입) {  
    // ...  
}
```

매개변수와 같이 this의 타입을 적어주지만 실제로 인자값을 받는 매개변수는 `this: 타입`을 제외한 나머지임으로 헷갈리면 안된다.

예제 11-11

```
interface User {  
    name: string,  
    age: number,  
    init(this: User): () => {};  
}  
  
let user1: User = {  
    name: 'mjo',  
    age: 20,  
    init: function(this: User) {  
        return () => {  
            return this.age;  
        }  
    }  
}  
  
let getAge = user1.init();  
let age = getAge();  
console.log(age); // return: 20
```

11.7 콜백에서의 this

콜백 함수에서 `this` 는 콜백으로 함수가 전달되었을 때 `this` 를 구분해줘야 할 때가 있다.

| 예제 11-12

```
interface BrowserEL {
    addClickListener(onclick: (this: void, e: Event) => void): void;
}

class Handler {
    info: string;
    onClick(this: Handler, e: Event) {
        //BrowserEL의 this타입은 void인데 Handler라고 타입선언하여 에러
        this.info = e.message;
    }
}
let handler = new Handler();
browserEL.addClickListener(handler.onClick); //Error!
```

만약 `BrowserEL` 인터페이스에 맞춰 `Handler` 를 구현하려면 아래와 같이 변경해야한다.

| 예제 11-13

```
class Handler {
    info: string;
    onClick(this: void, e: Event) {
        //this의 타입이 void이기 때문에 여기서 this.변수명 사용 불가
        console.log('clicked!');
    }
}
let handler = new Handler();
browserEL.addClickListener(handler.onClick);
```

11.8 함수 오버로드

JavaScript는 동일한 매개변수지만 타입을 다르게 받을 수 있다. TypeScript에서 그 특징을 구현하기 위해 함수 위에 매개변수의 다른 타입을 적어둘 수 있다. 전달받은 매개변수의 개수나 타입에 따라 다른 동작을 하게하는 것을 오버로드라 한다.

11.8.1 매개변수의 개수는 동일하지만, 타입이 다른 경우

첫 번째 함수의 매개변수는 모두 `string` 타입을 가지며, 두 번째 함수는 `number` 타입의 매개변수를 가진다. 즉, 반환하는 값이 `string` 일 수도 `number` 일 수도 있기 때문에 반환되는 자료형으로는 `any`를 선언한다.

| 예제 11-14

```
function sum(a: string, b: string): string;
function sum(a: number, b: number): number;

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

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

11.8.2 매개변수의 개수는 다르지만, 타입은 같은 경우

세 개의 함수 선언이 되어 있으며 각 함수는 타입은 같지만 인자를 받는 매개변수의 개수가 다르다. 첫 번째 함수는 매개변수가 1개, 두 번째 변수는 매개변수가 2개, 세 번째 변수는 매개변수가 3개이다.

매개변수의 개수가 다르기 때문에 선택적 매개변수 '?'를 사용하며, 반환 값은 '||'를 사용하여 null 또는 undefined인 경우 0을 반환한다.

| 예제 11-15

```
function sum(a: number) : number;
function sum(a: number, b: number): number;
function sum(a: number, b: number, c: number): number;

function sum(a: number, b?: number, c?: number): number{
    return a + (b || 0) + (c || 0);
}

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

11.8.3 매개변수의 개수와 타입이 다른 경우

| 예제 11-16

```
interface NicknameMaker{
    name: string,
    num: number,
    init(this: NicknameMaker): () => {};
}

function makeNickname(name: string, num: number | string): NicknameMaker | string {
    if(typeof num === "number"){
        return {
            name,
            num,
            init: function(this: NicknameMaker){
                return () => {
                    return this.name+this.num;
                }
            }
        };
    }else{
        return "이름 다음에는 숫자로 입력해주세요.";
    }
}

const getNickName= makeNickname("Mjo", 303).init(); //Error!
console.log(getNickName());
```

예제 11-16처럼 작성하면 값을 판별하기가 어렵다. 때문에 예제 11-17과 같이 작성해야 한다.

예제 11-17

```
interface NicknameMaker{
    name: string,
    num: number,
    init(this: NicknameMaker): () => {};
}

function makeNickname(name: string, num: number): NicknameMaker;
function makeNickname(name: string, num: string): string;
function makeNickname(name: string, num: number | string): NicknameMaker | string {
    if(typeof num === "number"){
        return {
            name,
            num,
            init: function(this: NicknameMaker){
                return () => {
                    return this.name + this.num;
                }
            }
        };
    }else{
        return "이름 다음에는 숫자로 입력해주세요.";
    }
}

const getNickName = makeNickname("Mjo", 303).init();
console.log(getNickName()); // result: Mjo303
```



12. 클래스

12.1 프로퍼티 & 메서드 선언

클래스의 프로퍼티, 메서드, 매개변수 선언 시 `: 타입` 형태의 **타입 주석(type annotation)**을 표기함으로써 타입을 명시한다.

| 예제 12-1

```
class Calendar {
    month: string;

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

    getMonth(): string {
        return this.month;
    }
}

const calendar = new Calendar("May");
console.log(calendar.getMonth()); //result: May
```

기본적으로 프로퍼티, 메서드, 매개변수의 타입을 표기하는 방식은 위와 같다. 그중에서 프로퍼티를 선언하는 방식에는 2가지가 있는데 한 번 알아보도록 하자.

12.1.1 가장 기본적인 프로퍼티 선언 방법

클래스 내에서 사용할 프로퍼티를 먼저 정의 및 선언하고 타입을 표기하는 방법이다.

특히 생성자 함수를 통해 초기화가 되는 프로퍼티의 경우, 초기화되기 전에 **프로퍼티 정의**가 꼭 필요하다. 정의가 없으면 에러가 나므로 주의해야 한다.

하지만 타입은 표기하지 않아도 타입 추론이 있기 때문에 에러가 발생하지 않는다.

예제 12-2

```
class Calendar {
    month1: string; // 생성자 함수로 초기화 되는 속성은 미리 정의해야 한다. 없으면 에러 발생.
    month2: string = 'Jun';

    constructor(month: string) {
        this.month1 = month;
    }
}

const calendar = new Calendar("May");
console.log(calendar.month1); //result: May
console.log(calendar.month2); //result: Jun
```

자바스크립트와의 차이점

```
class Calendar {
    month; // 있어도 되고, 없어도 된다.

    constructor(month) {
        this.month = month;
    }
}

const calendar = new Calendar("May");
console.log(calendar.month); //result: May
```

자바스크립트에서는 프로퍼티는 미리 선언해도 되고 하지 않아도 문제가 되지 않는다. 하지만 타입스크립트에서는 미리 정의하지 않으면 에러가 발생하므로 다시 한번 주의하자.

그러다 생길 수 있는 궁금증. 그렇다면 이렇게 선언하면 안 될까?

```
class Calendar {  
  
    constructor(month: string) {  
        this.month: string = month; // error!  
    }  
  
    getMonth(): string {  
        return this.month;  
    }  
}
```

일반적인 변수를 선언할 때 `const name: string = 'Maria'` 와 같이 선언이 가능하기 때문에 위의 경우에도 가능할 것처럼 보인다. 하지만 `this` 를 사용하여 프로퍼티를 선언하기 때문에 위와 같이 사용하는 것을 불가능하다.

12.1.2 생성자 매개변수를 이용한 프로퍼티 선언 방법

이 방법은 생성자 함수를 통해 초기화되는 프로퍼티의 경우 매개변수를 통해 선언하는 방법이다.

대신 프로퍼티명 앞에 **접근 제한자** or **readonly**를 표기해야만 선언이 가능하다.

| 예제 12-3

```
class Calendar {  
    constructor(private readonly month: string) {}  
  
    getMonth(): string {  
        return this.month;  
    }  
}
```

12.2 접근 제한자 (Access Modifier)

프로퍼티, 메서드, 매개변수 선언 시, 맨 앞에 접근 제한자를 표기함으로써 접근 범위를 제한 할 수 있다. `public`, `private`, `protected` 3가지의 키워드가 있으며, 접근 범위는 아래와 같다.

접근 제한자는 프로퍼티, 메서드, 매개변수의 가장 맨 앞에 표기한다. (다른 키워드랑 같이 사용 시에도 가장 맨 앞에 표기)

- `public` : 클래스 내부/외부 어디에서나 모두 접근할 수 있다. default값이므로 접근 제한자를 쓰지 않는 경우 `public`으로 선언된다.
- `private` : 해당 클래스 내부에서만 접근할 수 있다.
- `protected` : 해당 클래스 내부와 상속받은 클래스 내부에서만 접근할 수 있다.

보기 편하게 표로 정리하면 아래와 같다.

	<code>public (default)</code>	<code>private</code>	<code>protected</code>
해당 클래스 내부	O	O	O
해당 클래스 외부	O	X	X
상속받은 클래스 내부	O	X	O

예제 12-4

```
class Calendar {
    private month: string;

    public constructor(month: string) {
        this.month = month;
    }

    private getMonth(): string {
        return this.month;
    }
}

const calendar = new Calendar("May");
console.log(calendar.getMonth()); // error: Property 'getMonth' is private and only accessible within class 'Calendar'.
```

12.3 readonly 읽기 전용

`readonly` 는 프로퍼티 이름 앞에 표기함으로써 **값을 변경하지 못하도록 읽기 전용으로 설정해주는 기능**으로 `interface` 와 `type` 의 정의에서 사용할 수 있다.

이와 같은 기능은 함수의 매개변수가 전달받은 인자 값이 변경되면 안 되는 경우 적합하게 사용할 수 있으며, **의도치 않게 값이 변경됐을 때 발생할 수 있는 오류를 방지할 수 있다.**

| 예제 12-5

```
class Calendar {
    private readonly month: string;

    public constructor(month: string) {
        this.month = month;
    }

    private getMonth(): string {
        return this.month;
    }
}

const calendar = new Calendar("May");
console.log(calendar.month); // error : Property 'month' is private and only accessible
                            // within class 'Calendar'.
calendar.month = 'Jun' // error : Cannot assign to 'month' because it is a read-only pro
                        // perty.
```

12.4 extends 상속

`extends`는 상속받고자 하는 부모 클래스를 명시할 때 사용하는 키워드이다.

부모 클래스의 프로퍼티와 메소드를 그대로 받아오기 때문에 자식 클래스의 인스턴스에서 부모의 것을 자유롭게 사용 가능하다. (단, `private`는 사용 불가능.)

예제 12-6

```
class User {  
    public constructor(public name: string, public age: number) {}  
  
    public getName(): string {  
        return this.name;  
    }  
    public getAge(): number {  
        return this.age;  
    }  
}  
  
class newUser extends User {  
    public constructor(name: string, age: number) {  
        super(name, age);  
    }  
}  
let user1 = new newUser("영희", 20);  
console.log(user1.getName()); // result: 영희  
console.log(user1.getAge()); // result:20  
// public 이므로 클래스 외부에서 접근 가능
```

12.5 인터페이스(interface)

8장의 타입 별칭 & 인터페이스 파트에서 먼저 포함된 내용이며, 이곳에서 다시 한번 짚어보고자 한다.

- 우선 **인터페이스**는 **type**과 비슷하게 새로운 타입을 정의하는 또 다른 방법이다. 객체, 함수, 함수의 파라미터, 클래스 등에 사용할 수 있으며 상호 간에 정의한 약속 혹은 규칙을 의미한다.
- 인터페이스는 구현 없이 이름과 타입만 정의한 추상화된 프로퍼티나 메서드만 가질 수 있다.
- 클래스는 **implements** 키워드를 통해 미리 추상화된 **interface**를 채택하여 사용할 수 있다.
- 다중 상속이 가능하므로 인터페이스가 여러 개일 경우 예시와 같이 **쉼표()**로 나열할 수 있다.

ex) `class shape implements Circle, Triangle, Square`



여기서의 핵심 포인트!!

인터페이스로 정의한 프로퍼티나 메서드는 해당 클래스에 필수적으로 들어가야 하며, 오버라이딩해서 내용을 구현해야한다.

| 예제 12-7

```
interface User {
    name: string;
    age: number;
    getAge(age: number): void;
}

class newUser implements User {
    name: string = "철수";
    age: number = 20;
    getAge(age: number) {
        console.log(this.age);
    }
    constructor() {}
}
let user1 = new newUser();
user1; // result: {name: "철수", age: 20}
```

| 예제 12-8

```
interface User {
    name: string;
    age: number;
    getAge(age: number): void;
}

class newUser implements User {
    constructor(public name: string, public age: number) {}
    getAge(age: number) {
        console.log(this.age);
    }
}
let user1 = new newUser("영희", 20);
user1; // result: {name: "영희", age: 20}
```

12.6 추상 클래스

추상 클래스란 일반 메서드나 추상 메서드를 포함할 수 있는 클래스를 뜻하며, `class` 앞에 `abstract`라고 표기하여 정의한다.

추상 메서드란 구현된 내용없이 이름과 타입만 선언된 메서드를 뜻하며, 메소드명 앞에 `abstract`라고 표기한다.

추상 클래스는 상속용으로만 가능하기 때문에 일반 클래스와 달리 객체 인스턴스 생성이 불가능하며, 상속받은 클래스에서는 해당 추상 메서드를 반드시 구현(오버라이딩)해야한다.

| 예제 12-9

```
abstract class Circle {
    public abstract getArea(): number;

    public printArea(): string {
        return `${this.getArea()}`;
    }
}

class ShapeArea extends Circle {
    public constructor(protected readonly radius: number) {
        super();
    }

    public getArea(): number {
        return Math.PI * this.radius * this.radius;
    }
}

const myCircle = new ShapeArea(10);

console.log(myCircle.printArea()); // result: 314.15926535897
```

12.6.1 인터페이스vs 추상 클래스

둘 다 추상적인 메서드를 정의한다는 공통점이 있다. 하지만 인터페이스는 일반 메서드를 포함 할 수 없고, 추상 클래스는 일반 메서드를 포함 할 수 있다는 점에서 차이를 갖는다.

	인터페이스	추상 클래스
추상 메서드	O	O
일반 메서드	X	O



13. 제네릭 기초

13.1 제네릭이란 무엇일까?

| 예제 13-1

```
function helloString(message: string): string {
    return message;
}
```

위 함수는 인자로 `string` 타입의 `message`를 받으며 `return` 타입도 `string`이다.

| 예제 13-2

```
function helloNumber(message: number): number {
    return message;
}
```

위 함수는 인자로 `number` 타입의 `message`를 받으며 `return` 타입도 `number`이다.

위 함수들은 각각 일정한 타입을 인자로 사용하고 `return`으로 받으며 반복되고 있다.
그리고 더 많은 반복이 일어날 수 있고, 반복적인 함수가 생길 수 있다.

그래서 우리는 모든 타입을 받을 수 있고 `return` 할 수 있는 `any`를 사용했다.
하지만 `any`를 사용하게 되면 우리의 의도와는 다르게 다른 결과를 가져온다.

예제 13-3

```
function hello(message: any): any {
    return message;
}
console.log(hello("hey"));
console.log(hello(30));

console.log(hello("hey").length); // result: 3
console.log(hello(30).length); // result: undefined
```

위 함수는 `any` 를 사용한 범용적인 함수이다.

13.2 제네릭 기초

그렇다면 제네릭을 어떻게 사용하는지 한 번 알아보자.

홀 화살 팔호를 사용하고 타입의 `T` 를 사용해 아래와 같이 사용한다.

예제 13-4

```
function helloGeneric<T>(message: T)
console.log(helloGeneric("hey"));
```

`message` 라고 받은 인자를 `T` 라고 지정한다. 그렇게 되면 `helloGeneric` 이라는 함수 내에서 `T` 라고 하는 것을 기억하게 된다.

그러고 나서 예를 들어, 위 콘솔에 문자열을 넣게 되면, `T` 가 문자열로 지정되게 된다.

예제13-5

```
function helloGeneric<T>(message: T): T {
    return message;
}
console.log(helloGeneric('hey'));
// result: hey

// 위 함수는 다음과 같다.
// function helloGeneric<"hey">(message: "hey"): "hey"
```

그리고 `return` 값을 `T`로 지정하면 된다. 즉, `T`의 타입에 따라 함수의 타입이 결정된다.

`console.log()`을 찍어보자. 그러면 `helloGeneric`이라는 함수는 인자가 `string`이기에 리터럴 타입을 확인한 후,

`string` 리터럴 타입을 가지는 함수가 된다.

예제13-6

```
console.log(helloGeneric('hey').length);
// (property) String.length: number
// Returns the length of a String object.

console.log(helloGeneric(30).length);
// any
// result: Property 'length' does not exist on type '30'.
```

`length`은 문자열의 길이 즉, `number` 타입을 반환하기 때문에 콘솔을 확인하면 `number`를 반환하는 것을 알 수 있다. 또한, 숫자를 넣으려고 시도하면, 자동완성기능에 `number` 메서드를 확인하고 `length`를 인식하지 못하는 것을 알 수 있다.

예제 13-7

```
console.log(helloGeneric(true));
// function helloGeneric<true>(message: true): true
```

이번엔 **boolean** 타입을 대입해 보자.

인자가 **boolean**으로 들어가고 그에 따라 **T**의 타입이 **boolean**, **return**값 또한 **boolean**이 된다.

제네릭의 장점은 **T**를 변수처럼 사용해 **Type**을 지정해 줄 수 있다.

any는 모든 것을 반환하고 제네릭은 지정해서 사용하는 차이점을 가진다.

any는 들어오는 **input**에 따라 달라지는 타이핑을 할 수 없지만 제네릭은 가능하다.

함수를 만들어보자.

예제 13-8

```
function helloGeneric<T, U, K>
```

홀 화살 괄호 내에는 T뿐만 아니라 2개, 여러 개를 포함시킬 수 있다.

T, U, K는 함수 내에서 유효한 제네릭이다.

작성하는 방법은 **클래스**, **array**, **함수** 등 더 많아질 수 있으나 사용하는 방법은 명확하다.

예제 13-9

```
function helloBasic<T>(message: T): T{
    return message;
}
helloBasic<string>("Hey");
// function helloBasic<string>(message: string): string
helloBasic(36);
```

위 함수에서 제네릭을 가져다 써서 **변수처럼 지정해 쓰는 방식과 그렇지 않은 방식**이 있다.

위와 같이 제네릭을 쓰지 않으면 `<T>`가 자동적으로 추론이 된다. 추론 규정에 따라서 `T`가 `36`이 된다.

우리의 생각으로는 `36`이면 `number`로 출력이 되어야 한다고 생각하겠지만 TS는 가장 좁은 범위의 타이핑을 추론하기 때문에 `36`을 넣게 되면 `T` 자체는 `36`이 된다. 결과물인 `return type`도 `36`이 된다.

그러니, 넣어서 사용하면 `<T>`뒤의 인자가 제한이 되고, 넣지 않고 사용하면 `<T>`가 추론된다고 이해하면 된다.

그렇다면, 함수를 만들어보자.

| 예제 13-10

```
function helloGeneric<T>(message: T): T{
    return message;
}

helloGeneric<string>("Hey");
// function helloGeneric<string>(message: string): string
helloGeneric<string>(39); // result: Error
```

`string`으로 타입을 지정했으니, `39`라는 숫자를 넣으면 에러가 뜬다. (`string`으로 지정된다.)

| 예제 13-11

```
helloGeneric(39);
// function helloGeneric<39>(message: 39): 39
```

(자동으로 추론되어 `39`으로 지정된다.)

예제 13-12

```
function helloGeneric<T, U>(message:T, comment:U):T{
    return message;
}

helloGeneric<string>("Hey"); // result: Expected 2 type arguments, but got 1.
helloGeneric(36); // result: Expected 2 type arguments, but got 1.

helloGeneric<string, number>("Hey", 12);
// function helloGeneric<string, number>(message: string, comment: number): string
helloGeneric(21, 31);
// function helloGeneric<21, number>(message: 21, comment: number): 21
```

또한, 위 예제와 같이 `T` 뿐만 아니라 더 많은 인자를 넣어줄 수 있다.

현재, `U`를 `return` 태입에 사용하지 않고 있기에 의미가 없긴 하지만 `T` 와 조합해서 사용하면 의미가 있을 수 있다.

위와 같이 두 가지 **제네릭**을 사용하는 경우엔 인자를 `U` 까지 채워라는 에러 메시지가 발생하는 것을 볼 수 있다.

`string, number`로 `T, U`를 채워 주었고, 인자로 `"Hey"` 와 `12`를 넣어주니 에러가 사라졌다. 또한 변수를 지정하지 않은 방식 또한 `21, 31`을 넣어줌으로써 `T, U`를 `21, 31`로 만들어 주었다.



14. 유ти리티 타입

 유ти리티 타입은 정의해 놓은 타입을 변환할 때 사용하기 좋은 TypeScript가 제공하는 도구이다. 유ти리티 타입을 쓰지 않더라도 기본 문법으로 타입을 변환할 수 있지만 유ти리티 타입을 사용하면 좀 더 간결하게 타입을 변환할 수 있다.

14.1 Partial<Type>

Type 의 모든 프로퍼티를 선택적으로 타입을 변환하는 유ти리티 타입이다.

타입을 설정할 때 선택적으로 설정하지 않는다면 예제 14-1에서 author와 publisher가 없다고 에러가 뜬다.

| 예제 14-1

```
interface Book {
    title: string;
    description: string;
    author: string;
    publisher: string;
}

let typescript: Book = {
    title: "일잘딱깔센 TypeScript",
    description: "타입스크립트 입문자가 읽으면 좋은 책",
}; // 불가능
```

이 때 `Partial<Type>` 을 사용하면 예제 14-2 처럼 Book의 모든 프로퍼티를 선택적으로 바꿔줌으로서 에러를 잡을 수 있다.

예제 14-2

```
interface Book {
    title?: string;
    description?: string;
    author?: string;
    publisher?: string;
}

let typescript: Partial<Book> = {
    title: "알잘딱깔센 TypeScript",
    description: "타입스크립트 입문자가 읽으면 좋은 책",
}; // 가능
```

14.2 Required<Type>

`Required<Type>` 은 `Partial<Type>` 과 반대의 개념이다.

`Partial<Type>` 이 `Type` 의 모든 프로퍼티를 선택적으로 타입을 변환하는 유ти리티 타입이라면, `Required<Type>` 은 `Type` 의 모든 프로퍼티를 필수로 설정한 타입을 생성하는 유ти리티 타입이다.

예제 14-3에서 `author`와 `publisher` 프로퍼티를 선택적으로 바꿔주면서 에러가 발생하지 않는다.

예제 14-3

```
interface Book {
    title: string;
    description: string;
    author?: string;
    publisher?: string;
}

let typescript: Book = {
    title: "알잘딱깔센 TypeScript",
    description: "타입스크립트 입문자가 읽으면 좋은 책",
}; // 가능
```

하지만 `Required<Type>` 유ти리티 타입을 사용한다면 예제 14-4 처럼 모든 프로퍼티를 필수로 바꾸기 때문에 `author`와 `publisher`가 없다고 에러가 뜬다.

| 예제 14-4

```
interface Book {
    title: string;
    description: string;
    author: string;
    publisher: string;
}

let typescript: Required<Book> = {
    title: "알잘딱깔센 TypeScript",
    description: "타입스크립트 입문자가 읽으면 좋은 책",
}; // 불가능
```

14.3 Readonly<Type>

Type 의 모든 프로퍼티를 재할당이 불가능한 읽기 전용(Readonly) 으로 설정한 타입을 생성한다.
읽기 전용(Readonly)가 아니면 재할당이 가능하다.

| 예제 14-5

```
interface Book {
    title: string;
    description: string;
    author?: string;
    publisher?: string;
}

let typescript: Book = {
    title: "알잘딱깔센 TypeScript",
    description: "타입스크립트 입문자가 읽으면 좋은 책",
};

typescript.title = "TypeScript Basic"; // 재할당 가능
```

하지만 Readonly<Type> 을 준다면 예제 14-6처럼 모든 프로퍼티가 읽기 전용(Readonly) 이 되면서 재할당이 불가능하게 된다.

예제 14-6

```

interface Book {
  readonly title: string;
  readonly description: string;
  readonly author?: string;
  readonly publisher?: string;
}

let typescript: Readonly<Book> = {
  title: "알잘딱깔센 TypeScript",
  description: "타입스크립트 입문자가 읽으면 좋은 책",
};

typescript.title = "TypeScript Basic"; // 재할당 불가능

```

14.4 Record<Keys, Type>

프로퍼티 타입을 `Keys` 로, value 타입을 `Type` 으로 지정해 생성한다. `Record<Keys, Type>` 유ти리티 타입은 프로퍼티를 다른 타입으로 매핑하고 싶을 때 사용하면 좋다.

예제 14-7

```

interface Food {
  "1팀": "피자" | "치킨" | "햄버거" | "컵라면";
  "2팀": "피자" | "치킨" | "햄버거" | "컵라면";
  "3팀": "피자" | "치킨" | "햄버거" | "컵라면";
  "4팀": "피자" | "치킨" | "햄버거" | "컵라면";
}

const food: Food = {
  "1팀": "피자",
  "2팀": "치킨",
  "3팀": "햄버거",
  "4팀": "컵라면",
};

```

예제 14-7을 보면 value 값의 반복이 보인다. 이를 `Record<Keys, Type>` 을 활용하면 해결할 수 있다.

예제 14-8

```
const food: Record<
    "1팀" | "2팀" | "3팀" | "4팀",
    "피자" | "치킨" | "햄버거" | "컵라면"
> = {
    "1팀": "피자",
    "2팀": "치킨",
    "3팀": "햄버거",
    "4팀": "컵라면",
};
```

훨씬 간결해진 모습이지만 조금 길어 보인다. 이를 `type` 을 활용해 정리해 줄 수 있다.

예제 14-9

```
type Team = "1팀" | "2팀" | "3팀" | "4팀";
type Food = "피자" | "치킨" | "햄버거" | "컵라면";

const food: Record<Team, Food> = {
    "1팀": "피자",
    "2팀": "치킨",
    "3팀": "햄버거",
    "4팀": "컵라면",
};
```

`type` 을 활용해 정리해 주면 재사용에 용이하다.

14.5 Pick<Type, Keys>

Type에서 프로퍼티 Keys를 Pick(선택)해 타입을 생성한다.

아래 예제는 Person에서 name과 age를 선택해 kim을 생성한다.

| 예제 14-10

```
interface Person {
  name: string;
  age: number;
  location: string;
  gender: "M" | "W";
}

const kim: Pick<Person, "name" | "age"> = {
  name: "Kim",
  age: 27,
};
```

만약 name과 age가 아닌 다른 key를 선택하게 되면 에러가 발생한다.

| 예제 14-11

```
interface Person {
  name: string;
  age: number;
  location: string;
  gender: "M" | "W";
}

const kim: Pick<Person, "name" | "age"> = {
  name: "Kim",
  location: "Jeju", // Error: Type '{ name: string; location: string; }' is not assignable to type 'Pick<Person, "name" | "age">'
};
```

Pick<Type, Keys> 유ти리티 역시 type을 사용해 정리할 수 있다.

예제 14-12

```
interface Person {
  name: string;
  age: number;
  location: string;
  gender: "M" | "W";
}

type Kim = Pick<Person, "name" | "age">;

const kim: Kim = {
  name: "Kim",
  age: 27,
};
```

14.5 Omit<Type, Keys>

Pick 유ти리티 타입과는 반대 개념으로

Type에서 프로퍼티 Keys를 Omit(생략)해 타입을 생성한다.

아래 예제는 Person에서 age와 gender를 생략해 kim을 생성한다.

예제 14-13

```
interface Person {
  name: string;
  age: number;
  location: string;
  gender: "M" | "W";
}

const kim: Omit<Person, "age" | "gender"> = {
  name: "Kim",
  location: "Jeju",
};
```

만약 age나 gender를 포함하게 되면 에러가 발생한다.

예제 14-14

```
interface Person {
  name: string;
  age: number;
  location: string;
  gender: "M" | "W";
}

const kim: Omit<Person, "age" | "gender"> = {
  name: "Kim",
  location: "Jeju",
  gender: "M", // Error: Type '{ name: string; location: string; gender: string; }' is not assignable to type 'Omit<Person, "age" | "gender">'};
};
```

Omit<Type, Keys> 유ти리티 역시 type 을 사용해 정리할 수 있다.

예제 14-15

```
interface Person {
  name: string;
  age: number;
  location: string;
  gender: "M" | "W";
}

type Kim = Omit<Person, "age" | "gender">;

const kim: Kim = {
  name: "Kim",
  location: "Jeju",
};
```

14.5 Exclude<Type, ExcludeUnion>

`ExcludeUnion`에 들어갈 수 있는 모든 유니온을 `Type`에서 제외한 타입을 생성한다.

| 예제 14-16

```
type T1 = Exclude<"kim" | "lee" | "park", "park">;
// result : type T1 = "kim" | "lee"

type T2 = Exclude<string | number | boolean, number | boolean>;
// result : type T2 = string
```

예제 14-15를 보면 `type` T1에서 "park"을 제외한 "kim" | "lee"가 남게 된다.

`type` T2 역시 number와 boolean을 제외한 string이 남게 된다.

| 예제 14-17

```
type T1 = Exclude<string | number | boolean, number>;
// result : type T1 = string | boolean

type T2 = Exclude<T1, boolean>;
// result : type T2 = string
```

예제 14-16처럼 사용할 수도 있다.

14.6 NonNullable<Type>

`Type`에서 `null`과 `undefined`를 제외하고 타입을 생성한다.

| 예제 14-18

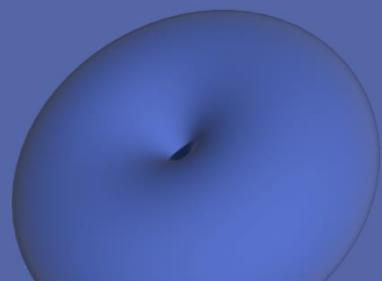
```
type T1 = NonNullable<string | number | boolean | null | undefined>;
// result : type T1 = string | number | boolean
```

T1에서 null과 undefined를 제외한 string과 number, boolean이 남게 된다.

03 마치며

참고자료

맺음말





참고자료

참고자료	URL	작성자
w3school	https://www.w3schools.com/typescript/typescript_getstarted.php	집필자 전원
타입스크립트 핸드북	https://joshua1988.github.io/ts/intro.html	집필자 전원
땅콩코딩	https://www.youtube.com/c/땅콩코딩	박성범
알잘딱깔센 Github	https://paullabworkspace.notion.site/GitHub-435ec8074bcf4353afb947f601a030dfd	박성범
알잘딱깔센 JavaScript	https://paullabworkspace.notion.site/JavaScript-f037c206e538471f9a9f1915b2139a60	박성범
타입스크립트 공식문서	https://www.typescriptlang.org/ko/docs/handbook/intro.html	김민석
타입스크립트 깃북	https://typescript-kr.github.io/	김민석
타입스크립트 퀴즈스타트	https://www.rubypaper.co.kr/76	김성훈
타입스크립트 프로그래밍	http://ebook.insightbook.co.kr/book/122	주다빈
Do it! 타입스크립트 프로그래밍	http://www.easyspub.co.kr/20_Menu/BookView/434/PUB	주다빈
코딩왕ma TypeScript #4 함수 - 타입스크립트 강좌	https://www.youtube.com/watch?v=prfgfj03_VA	조민경
[블로그] 함수 오버로딩이란?	https://developer-talk.tistory.com/307	조민경
코딩왕ma TypeScript #5 리터럴, 유니온/교차 타입 - 타입스크립트 강좌	https://youtu.be/QZ8TRIJWCGQ	전유진



코딩애플 타ypeScript 쓰는 이유 & 필수 문법 10분 정리	https://www.youtube.com/watch?v=xkpcNolC270	전유진
TypeScript 가이드북	https://yamoo9.gitbook.io/typescript/	이세영
[블로그] 클래스	https://pannchat.tistory.com/entry/TypeScript-클래스-추상클래스-인터페이스-그리고-차이	이세영
기초부터 블록체인 실습까지 단숨에 배우는 타ypeScript	http://www.yes24.com/Product/Goods/102416447	양다은



맺음말

앞선 내용을 통하여 우리는 타입스크립트의 기초 문법을 다루었습니다. 이 책을 읽고 나신 여러분은 아직 타입스크립트를 실무에서 어떻게 사용해야 할지 잘 모를 수 있습니다. 하지만 적어도 타입스크립트란 무엇이며 어떤 특징을 가졌는지 알게 됐을 것입니다. 이 책의 목적은 타입스크립트를 공부하기 시작하기에 앞서 일종의 마중물 역할을 하는 것입니다.

사실 대부분의 주니어 개발자들에게 타입스크립트라는 기술 스택은 취업시장에서 유리할 수 있지만 필수는 아닙니다. 미래의 프론트엔드 개발자라면 타입스크립트는 필수이지만 주니어 개발자들에게 무엇보다 중요한 건 자바스크립트와 리액트 기본기입니다. 따라서 이 책을 통하여 타입스크립트에 대한 막연함을 해결하고 나중에 더 깊은 공부를 이어 나가시기를 추천해 드립니다.

본 전자책을 집필한 우리 주니어 개발자 10인은 이제 각자의 꿈과 목표를 가지고 더 나은 개발자가 되기 위하여 앞으로 한 발짝 내딛습니다. 책을 집필하며 많이 부족한 점도 있었지만, 모르는 부분을 채우기 위해 공부를 하고 인생 첫 집필에 도전하며 한 단계 성장한 우리 자신을 느낍니다.

기회를 잡고 실수를 하라. 그것이 당신이 성장하는 방법이다.

_Mary Tyler Moore

끝으로 본 집필을 처음부터 끝까지 도와주신 위니브 이호준 대표님께 감사의 인사를 드립니다. 그리고 각자의 꿈과 목표를 위해 도전을 하고 실패도 하고 다시 일어서며 성장하시는 여러분, 여러분의 꿈과 목표가 이루어지길 진심으로 응원합니다.

감사합니다.

제주코딩베이스캠프 로드맵



제주코딩베이스캠프 온라인 강의

<https://bit.ly/3Jf8NIH>

초판 1쇄 발행 | 2022. 06. 02

지은이 | 김민석, 김민영, 김성훈, 박성범, 양다은, 이세영, 임현지, 전유진, 조민경, 주다빈

편집 | 김진, 차경림

총괄 | 이호준

펴낸곳 | 사도출판

주소 | 제주특별자치도 제주시 동광로 137 대동빌딩 4층

표지디자인 | 차경림

홈페이지 | <http://www.paullab.co.kr>

E-mail | paul-lab@naver.com

ISBN | 979-11-88786-60-2 (PDF)

Copy right © 2022 by. 사도출판

이 책의 저작권은 사도출판에 있습니다. 저작권법에 의해 보호를 받는 저작물이므로 무단 복제 및 무단 전재를 금합니다.

이 책에 대한 의견을 주시거나 오탈자 및 잘못된 내용의 수정 정보는 사도출판의 이메일로 연락을 주시기 바랍니다.

“**알아서 잘**
“**똑같은 꿈하고**
센스있게
정리하는
TypeScript
핵심 개념”

비매품/우료

95560



9791188786602

ISBN 979-11-88786-60-2 (PDF)

