

# Ver.2021 노베이스 모던 웹개발

- React 시작 전에 알아야 할 JavaScript, TypeScript

작성자: 조교행님

## 목차

p.2	-	0. Orientation
p.9	-	1. 주석(Comment)
p.11	-	2. 변수(Variable)
p.17	-	3. 자료형(Data Type)과 백틱(``), 비교연산자
p.21	-	4. if, else,   , &&, ?, :
p.29	-	5. 배열(array)
p.34	-	6. 객체(object)
p.39	-	tip. 객체와 배열에 대한 깊은 이해
p.45	-	7. 함수(function) 기본이론 1 - with TS
p.53	-	8. 함수(function) 기본이론 2 - with TS
p.64	-	9. 화살표 함수(arrow function)
p.69	-	10. 반복 - map() 과 filter()
p.72	-	11. interface - TS only
p.79	-	12. 마치며

## 0. Orientation

ver.2021 노베이스 모던 웹개발 코스의 첫번째 시간에 온 걸 환영한다. 이 코스에선, 다음 기술로 웹앱(브라우저로 작동하는 프로그램)을 만들어볼 것이다.

1. TypeScript
2. React Hook
3. React Router
4. Apollo
5. Tailwind CSS
- 
6. GraphQL Backend
7. SQLite

언급한 기술들은 2021년 기준으로 모던 웹, 즉 최신 웹 기술들이다. 먼저 다음 질문에 답해보겠다.

Q. 진짜 노베이스 수업 맞나?

A. 저 리스트를 처음 보는 순간 이게 대체 뭔가 싶을 것이다. 잘못 찾아왔다는 생각에 벌써 도망갈 생각 하고 있는지도 모른다. 그러나, 이 코스는 정말로, 프로그래밍의 기초인 변수, 자료형, 배열, 조건문, 함수부터 차근차근히 배워가는 노베이스 코스가 맞다.

Q. 왜 하필 저 기술들인가?

A. 글을 쓰는 현재 시점에서 가장 쉽게, 눈에 보이는 프로그램을 만들 수 있는 기술이기 때문이다. 컴퓨터공학을 전공한 학생들조차도, 백준 알고리즘 문제는 엄청 잘 풀면서, 졸업할때까지 자신의 힘으로 프로그램 하나 만들 줄 모르는 학생들이 생각보다 많다는 건 안타까운 현실이다. 전공자든 비전공자든 일단 뭔가 만들어봐야하지 않겠는가?

Q. 난이도는 둘째치고 배워야 할 과목이 무려 7개나 되지 않나?

A. 7개 각각이 하나의 과목이라 생각하면 당연히 막막하다. 저 7개가 각각 의미하는 것은 '과목'이 아니라 '기술'이다. 해당 기술을 깊게 들어가면 한도 끝도 없이 어렵고 그것 하나만으로도 책이 되지만, 우리 개발할 때 쓰는 것들만 간단히 배워볼 것이다.

Q. 웹 시장이 너무나 빨리 변하기 때문에, 1년후에 구식이 될지도 모르는 기술들이다. 배우는 게 의미가 있을까?

A. 그런 생각이라면 아무것도 만들 수 없다. 신기술은 끊임없이 나오기때문에, 우리 새로운 기술을 끊임없이 기다리는게 아니라, 이미 나와있는 기술들을 사용해봐야된다. 당연히 저 기술들은 언젠가 구식이 될 것이다. 우리 그런 거 따질 게 아니라, 어느 정도 배웠을 때 잠시 멈춰 스스로 자신만의 프로그램을 만들어봐야한다.

이 수업을 듣기 위해 필요한 지식은 아무것도 없다. **컴퓨터를 켜서 구글 크롬에 접속할줄만**

**알면 바로 시작해도 된다.** 만약, 수업을 듣다가 모르는 단어나 기술이 나온다면, 설명을 빼먹은 게 아니라 지금은 설명할 단계가 아니라는 필자의 판단 때문에 설명을 하지 않은 것이다. 그냥 책 읽는다 생각하고 쪽쪽 앞으로 나가자. 다시 말하지만 이 수업의 대상은 '노베이스', 컴퓨터로 할 줄 아는 건 게임밖에 없는 학생들을 위한 것이다.

지금부터, 모던 웹의 기본이라고 할 수 있는 JavaScript, TypeScript 수업을 시작한다.

\* 수업에서, 필자는 학생들이 영어로 꼭 알아줬으면 하는 단어는 영어로 썼다. 프로그래머는 좀 힘들더라도, 영어에 익숙해져야한다. 우리가 마주칠 기술들의 공식문서는 영어로 쓰여져 있고, 버그(= 에러)를 잡기 위해 구글링을 할 때도 영어로 찾는 게 훨씬 좋다. 일부러 영어로 쓴 단어는 개발을 위해 꼭 알아줬으면 하는 단어이기에, 꼭 알아두도록 하자!

JavaScript: 웹에 쓰이는 단 하나의 프로그래밍 언어이다. 하나밖에 없다는 건, 불편할 순 있어도 어쩔 수 없이 써야되는 것이다.

\* 프로그래밍 언어: 컴퓨터와 소통하기 위한 언어이다. 요즘은 google assistant, siri, alexa 처럼, 뭐 좀 해달라하면 알아서 척척 해주는 똑똑한 것들이 생겼지만, 그것들조차 조금만 복잡하게 말하면 무슨 말인지 못알아듣는다. 컴퓨터와 대화하기 위해선, 대충 말해도 알아듣는 사람의 언어와는 다르게, 정해진 프로그래밍 언어의 문법대로 똑바로 말해줘야한다. 전 세계에는 한국어, 영어, 일본어, 중국어 등등 약 6,500개의 언어가 있듯이, 프로그래밍 언어는 지금까지 약 700개가 만들어졌다.

만약 브라우저를 떠나서, 서버 개발자의 세계만 가도 수많은 기술들이 있어 선택의 자유가 있지만, 브라우저는 오로지 JavaScript만 프로그래밍 언어로 인정한다. 왜 그랬을까? 그렇게 하도록 정해놓았고, 안 바뀌었기 때문이다. 모든 브라우저는 오로지 JavaScript만 이해하며, 빠르게 변해가는 웹 시장의 수많은 제품들은 JavaScript로 작성되었다.

그러나 보통의 JavaScript개발자들은 개발 과정에서 버그 잡는데만 엄청난 시간을 쓴다. 특히, JavaScript는 Type이 없기 때문에 Type Check를 하지 않고, 데이터가 오면 그냥 받아들이며 각종 에러를 발생시켰다. 조금 어려운 문장이라 무슨 말인지 모를수도 있겠으니, 간단하게만 설명해보겠다. 이건 마치 중국집에 초밥 주문이 들어오는것과 같다. 주방장은 적어도 메뉴판에 있는 메뉴가 주문 올 것이라고 확신할텐데, JavaScript라는 카운터 직원은 중국집 직원인데도 초밥, 피자, 김치찌개 주문 거절을 못해 주방을 울스톱시킨다. 즉, 프로그램이 멈춰버린다. 이처럼 무슨 데이터가 들어올지 예상할 수 없는 프로그램은 당연히 위험할 수밖에 없다.



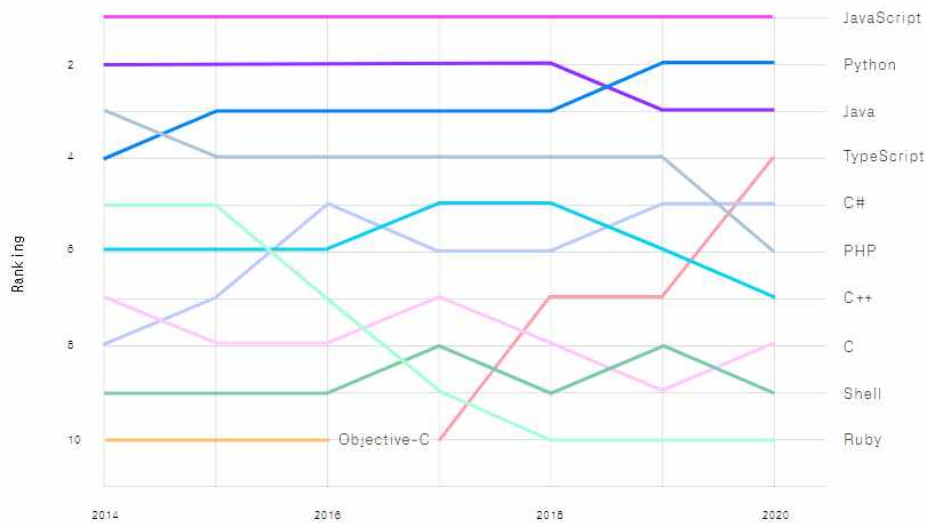
TypeScript(TS)는 이런 문제점을 보완하기 위해 등장했다. TypeScript는 너무나 자유로운 JavaScript에 '규칙'을 부여한다. 해당 규칙은 프로젝트 진행 과정에서 발생할 수 있는 버그들을 최소화할 수 있다. 이것은 우리가 작성하는 코드가 예측 가능하고, 읽기 쉬워진다는것을 의미한다. 메뉴판에 있는 주문만 주문받고, 만약 다른 걸 주문하면 주방에 오기 전에 전화로 그런 메뉴 없다고 안내하는 똑똑한 카운터 직원과 같다.

TypeScript를 써야 할 다른 이유가 있다. 마이크로소프트가 직접 개발했고, 관리하는 언어이기 때문이다. 웹 시장에서 대기업이 직접 해당 기술을 관리한다는건 그 언어가 매우 안정적이라는 증거이며, 기술을 선택할 때 매우 중요한 요소이다.

TypeScript는 JavaScript 다른 언어가 아니다. JavaScript에 '규칙'들이 더해진 것일 뿐이다. 쉽게 말하면 '업그레이드 버전'이며, 개선된 JavaScript이다. 따라서 JavaScript가 할 수 있는 거의 모든 것을 TypeScript로도 당연히 할 수 있다.

// The languages that dominated

## Top languages over the years



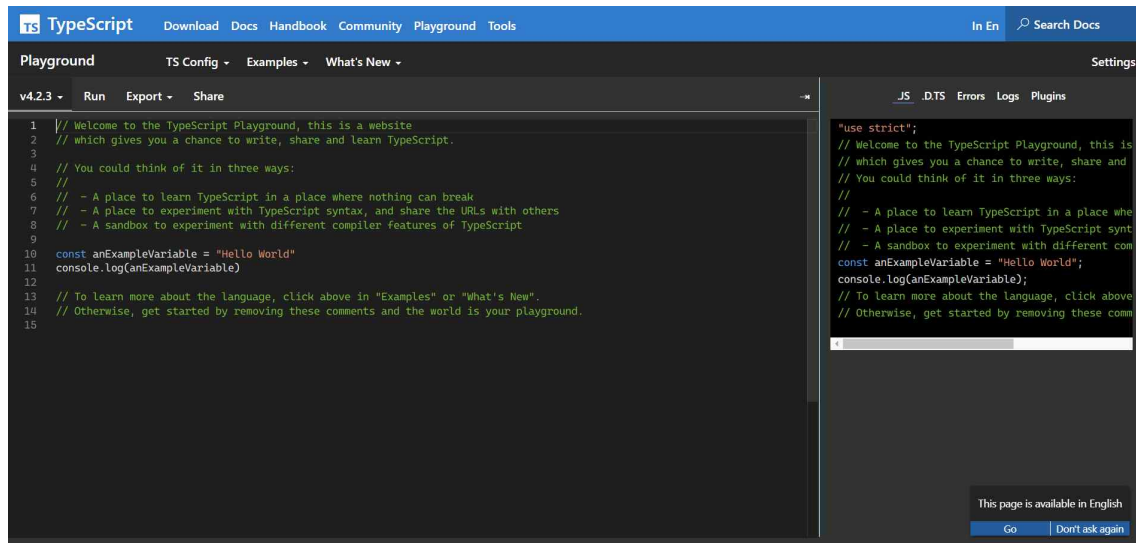
Github 연례보고서 2020 The State of the Octoverse.

TypeScript는 폭발적인 성장을 하고 있고, 이는 현재 진행형이다.

실습을 위해 별다른 개발환경세팅을 하진 않는다. 개발환경세팅을 경험해보는 건 매우 좋지만, 보통은 TypeScript만으로 개발하지 않고, 이 챕터의 목적은 단순히 JavaScript, TypeScript문법을 연습하는 것이기 때문에 개발환경세팅은 코스의 다음 강의인 React에서 해볼 것이다. 대신, Microsoft에서 제공하는 TypeScript Playground를 통해 JavaScript, TypeScript를 연습해보겠다.

PC의 크롬 브라우저로 다음 사이트에 접속하자.

<https://www.typescriptlang.org/play>

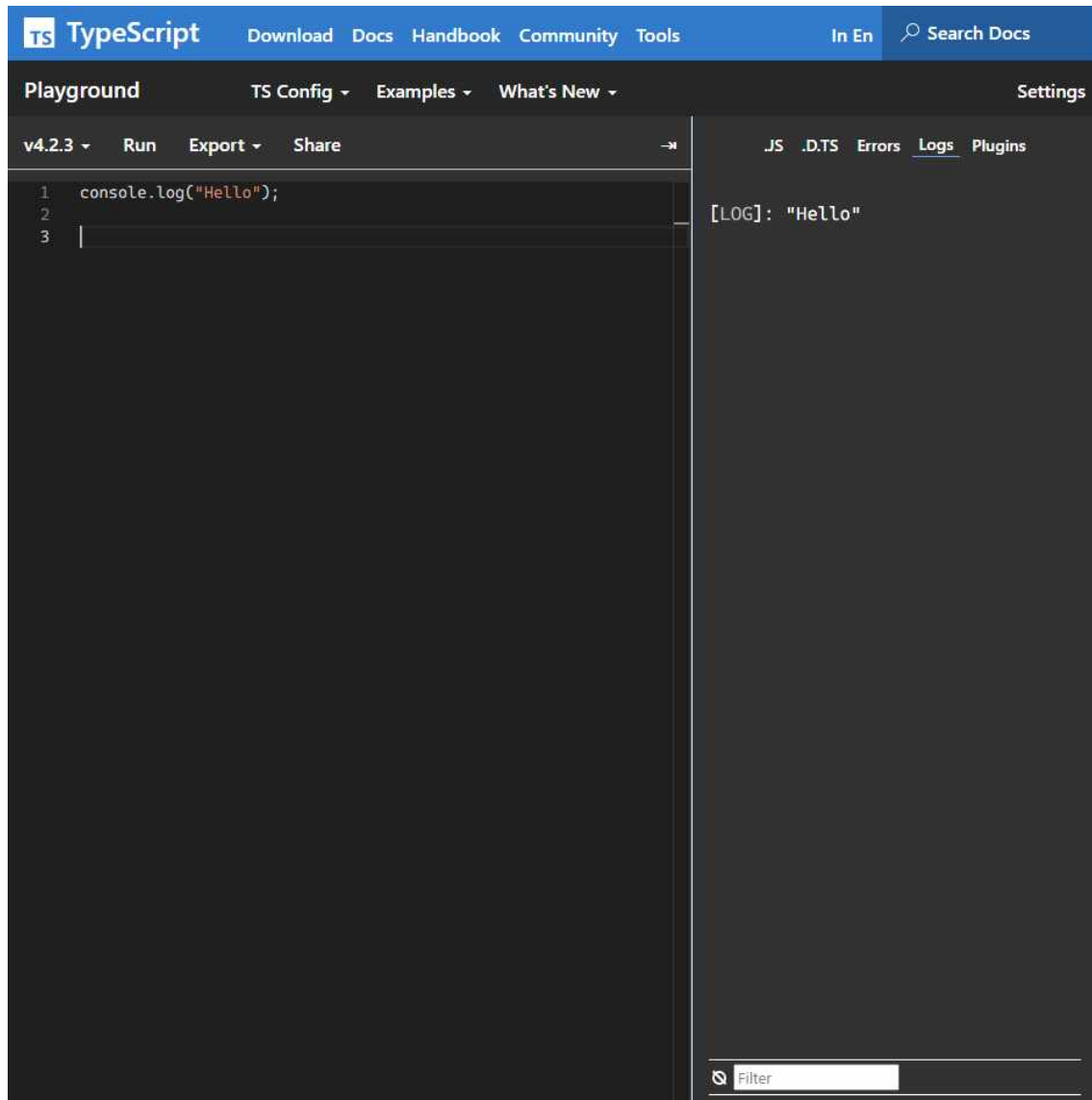


TypeScript Playground 화면. 현재 글을 쓰는 시점에서, TypeScript version은 4.2.3이다. 기존에 쓰여져 있는 모든 걸 삭제하고,

```
console.log("Hello");
```

라고 입력해보자.

오른쪽에 Logs 탭을 클릭하면 결과를 볼 수 있다. 코드를 입력하고 ctrl+enter 를 누르면 오른쪽 콘솔에 결과가 나온다. 이것을, 개발자들은 '출력'되었다라고 표현한다.



Hello가 정상적으로 나왔는가? 축하한다. 여러분의 첫 TypeScript 코드를 실행시켰다.

오른쪽 아래 동그라미에 대각선 선이 그어져있는 버튼을 클릭하면 콘솔창이 깨끗하게 지워진다.

- \* 콘솔(console): 윈도우에서 '명령 프롬프트(cmd)', 'PowerShell', 리눅스에서 'Terminal'이라고 부르는 것이다. 까만 배경에 명령어를 치면 실행한 결과를 보여준다.
- \* console.log() 는 TypeScript에서 콘솔 화면에 데이터를 찍어볼 때 쓰는, console 객체 안에 있는 함수다. 함수가 뭔지, 객체가 뭔지 몰라도, 지금은 일단 뭔가를 테스트할 때 쓰는 것이라고 알아두자.

배워볼 내용은 다음과 같다.

1. 주석(Comment)
2. 변수(Variable)
3. 자료형(Data Type)과 백틱(``), 비교연산자
4. if, else, ||, &&, ?, :
5. 배열(array)
6. 객체(object)
7. 함수(function) 기본이론 1 - with TS
8. 함수(function) 기본이론 2 - with TS
9. 화살표 함수(arrow function)
10. 반복 - map() 과 filter()
11. interface - TS only

TS가 따로 써져있는 부분을 제외하고, TypeScript와 JavaScript는 동일하게 작동한다. 즉, 이 강의는 6장까진 JavaScript 강의이며, 7장부터 TypeScript 에서만 작동하는 코드가 포함되어있다.

with TS 는, TypeScript 에서만 작동하는 코드가 포함되어 있는 장이다.

TS only 는, JavaScript 에선 동작하지 않고, TypeScript 에서만 동작한다.

TypeScript에 해당하는 부분은 파란색으로 작성한다. TypeScript에만 있는 개념은 저자가 꼭 TypeScript에만 해당된다고 설명할 것이다.

이제 가장 중요한, 주석(Comment) 부터 시작해보자.



## 1. 주석(Comment)

본래 책에서 중요한 것은 항상 앞부분에 나오고, 덜 중요할수록 뒷부분에 나온다. 주석을 맨 앞에 소개한 책은 잘 없을지 모르겠으나, 개발자로서 주석이 가장 중요하다 생각해 1장으로 선정했다.

주석(Comment) : **프로그램에 아무 영향을 끼치지 않는 부분**을 의미한다. 컴퓨터는 모르고 사람만 아는 것이다. 물론 용량은 차지한다. 주로 해당 **코드를 설명**할 때 쓴다. 코드를 직접 작성한 개발자나, 다른 개발자들이 해당 **코드를 읽기 쉽게** 해준다.

```
1  // 나는 오늘부터 TypeScript 공부를 시작했다.
2  // 즐겁다!
3
4  /*
5  여러줄 주석은
6  이렇게 사용한다.
7  */
8
9  const a = 1; // 1을 변수 a에 집어넣는다.
10
11 // 다음은 내가 처음 쓴 코드이다.
12 console.log('Hello World!');
```

주석은 // 를 주로 사용한다. // 부터, 해당 라인이 끝날때까지 주석이 적용된다. //쓰고 스페이스 한 칸 띄운 다음 주석을 쓴다.

여러 줄 주석을 쓸 땐 /\* \*/ 를 쓰기도 하는데, 사실 잘 안 쓴다. 여러 줄 쓰더라도 // 을 계속 쓰는 경우가 많다. 다른 개발자가 /\* \*/를 썼다면 주석이구나 하고 알기만 하면 된다.

주석을 쓸 때 신경써야될 건 다음과 같다.

line10: **코드 옆에 주석을 쓸 경우**, 이렇게 **칸을 띄워서 주석을 구분**한다.

line12: **위에 주석이 있으면 바로 아래 코드에 해당되는 주석**이다. 설명하고자 하는 **코드 아래에 주석을 쓰지마라**. 주석은 코드 위에 쓴다.

물론, 모든 코드에 일일이 주석을 달 필요는 전혀 없다. 간단한것도 다 주석을 붙이면 오히려 읽기 힘들다. 그러나 만약 코드가 좀 복잡하거나 어려워서, **일주일뒤에 내 코드를 볼 때 읽기 힘들 것 같을때는 반드시 주석을 달아야**한다. 전체 개발과정에서 70%를 차지하는 시간이, 만들어놓은 프로그램을 다시 고치고, 버그 잡고, 관리하는 '유지보수'에 걸린다.

이번 장은 이 내용이 전부이다! 이제 변수(Variable)를 배워보자.

핵심정리

주석을 옆에 쓸 경우: 칸을 띄워서 구분한다.

주석을 위에 쓸 경우: 아래 코드에 대한 설명이다.

주석은 코드 아래에 쓰지 않는다.

어렵거나 복잡하면 주석을 습관화하자.

## 2. 변수(Variable)

### 2-1. let

\* 지금부터, 읽다보면 '코딩 습관 들이기'라는 매우 지루할 수 있는 규칙 설명란이 나올 것이다. 사실 '코딩 습관 들이기' 부분은, **무시해도 프로그램은 잘 실행되는 것들만** 써냈다. 그래서 어려운 게 싫은 사람들은, 나올때마다 무시해도 좋으나, 좋은 습관을 미리 익혀놓고 싶은 사람들은 처음 배울때부터 습관을 들여놓도록 하자. 대형 앱(프로그램)을 만드는 고수 수준이 될수록 들여놓은 습관은 큰 도움이 된다.

컴퓨터는 기본적으로 계산기다. 페이스북같은 대형 웹앱도, 근본적으로는 끝없는 계산식으로 이루어져 있다.

정말 간단한 계산 하나 해보자.

```
1 console.log(2 + 3);
```

결과:

```
[LOG]: 5
```

코딩 습관 들이기:

연산자(+) 와 피연산자(2 그리고 3) 사이엔 칸을 한 칸 띄어준다.

어떤 하나의 문장을 끝마칠때는, 세미콜론(;)을 찍어준다.

실전상황에서 이 코드는 매우 쓸모없는 것이다.  $2 + 3$ 이 정해져있으면 그게 무슨 계산기인가? 적어도,  $a + b$  처럼,  $a$ 에 어떤 게 들어오든지,  $b$ 에 어떤게 들어오든지 둘을 더한 결과(+)를 낼 수 있어야 한다.

**변수(variable): 변경되거나, 변할 수 있는 수.** 우리는 쓰고싶은 변수에 이름을 지어줘야 한다. dog, cat, jajangmyeon으로 지어도 되지만, 나는 내가 쓰고자 하는 변수를  $a$ ,  $b$ 로 이름지어주겠다. **실전에선 이렇게, 원지 한번에 알아챌 수 없는 이름으로 지으면 안되지만,** 간단한 예시를 보여주기 위해  $a$ ,  $b$ 를 쓰겠다.

참고로, "짜장면" 이라고 **한글로 지으면 안된다.** 뭐가 되었든지, 영어로 써야한다.

```
1 let a
2 let b;
3 a = 1;
4 a = 2;
5 b = 3;
6 console.log(a);
7 console.log(b);
8 console.log(a + b);
```

결과:


```
[LOG]: 2
[LOG]: 3
[LOG]: 5
```

코딩 습관 들이기:

'=' 이 가운데에 있을 때는 한 칸 띄워준다.

이 그림을 설명하기 전에, 프로그래밍의 기초로서 확실히 익히고 넘어가야할 게 있다. 위 코드는 수학식처럼 보이지만 수학하던 방식으로 읽으면 안된다. 수학에선, 글을 읽듯이 왼쪽에서 오른쪽으로 읽는다. line3를 수학하던 방식으로 읽으면, 'a는 1'이다. 그러나, **프로그래밍에서 '=' 기호는 '입력' 을 의미하므로, 읽을때는 오른쪽에서 왼쪽으로 읽어야 한다.** 즉, **'1을 a에 집어넣는다'**로 이해해야 한다.

```
a = 1;
```



이제 변수를 이해해보자. 결과를 보면,

a는 2이고, b는 3이고, a + b는 5이다. a는 line3에서 1이었지만, line4에서 2로 **변했다.** 그래서 **변수다. 변경되거나, 변할 수 있는 수.**

let은 대체 뭐하는 놈일까? a라는 변수를 쓰기 위해선 컴퓨터에게 이걸 쓰겠다고 "선언"해야한다.

**선언은 컴퓨터에게 '나 이걸 쓰겠다' 고 말하는 행동**이다. 쓰고자 하는 변수 앞에 let을 붙여 '선언'한다. 그리고 선언 이후 그 변수를 '사용'할때는 let을 쓰지 않는다. line3, line4, line5 에서 변수를 사용하지만 let은 쓰지 않았다.

변수 선언하고, 값이 주어진다면, 그 변수는 '초기화(initialize. 줄여서 init)'되었다 라고 한다. line3 에서 a는 초기화가 완료된 것이다. **어떤 변수를 선언해 쓰고자 할 땐, 반드시 초기화까지 마쳐야한다.**

\* let 은 영어단어 let 에서 나왔다. '~하게 하다' 라는 의미다.

\* 선언은 명사로 declaration, 동사로 declare 라고 쓴다. 이 영어단어는 공개석상이나 연설, 정부기관에서 쓰는, 영어권 사용자들도 언론에서나 보는 어려운 단어다. 한국어로 번역된 '선언'이라는 단어도, 일상생활에선 잘 쓰지 않는다. 개발자들은 프로그래밍을 너무 많은 사람들이 배워서 자기들 밥줄이 끊기는 걸 염려해서인지, 가장 기본적인 개념을

설명하면서도 이런 어려운 단어를 쓰곤 한다.

위 코드는 다음과 같이 짧게 줄일 수 있다.

```
1  let a, b;  
2  a = 1;  
3  a = 2;  
4  b = 3;  
5  console.log(a, b, a + b);
```

결과:

```
[LOG]: 2, 3, 5
```

코딩 습관 들이기:

coma 뒤엔 한 칸 띄워준다.

line1: 여기서 a, b를 동시에 선언하기 위해 콤마로 구분지었다.

line5: console.log()에서 여러개를 출력시키고 싶을 땐 콤마를 쓴다.

선언과 동시에 값을 넣을수도 있다. 즉, 한 줄에 초기화까지 마칠 수 있다.

다음과 같이, a를 선언함과 동시에 0을 집어넣어 초기화시켰다.

```
1  let a = 0;  
2  a = 2;  
3  console.log(a);
```

결과:

```
[LOG]: 2
```

## 2-2. const

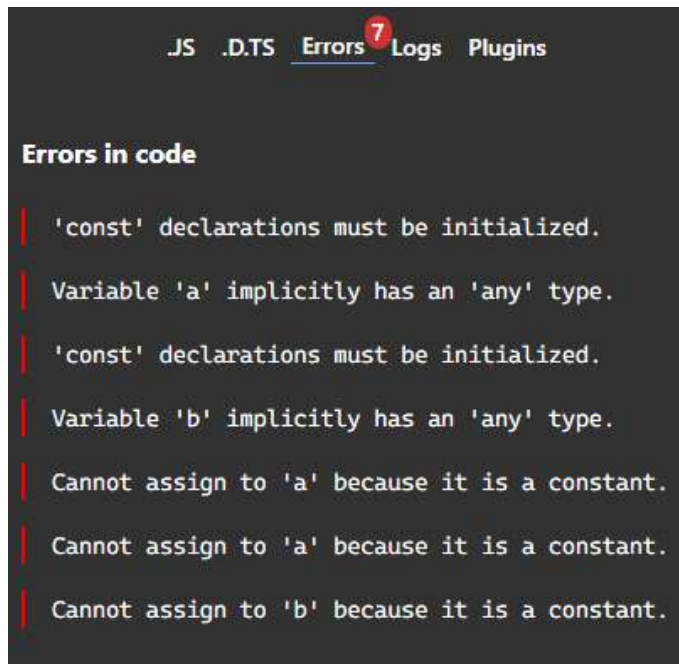
그러나, 지금까지 설명했던것과는 다르게, 우린 let을 사용하지 않고, 안정적인 프로그램을 위해 변수 선언의 대부분의 경우엔 const 를 사용할 것이다. const로 코딩하다가 안 돌아갈 때 let을 쓰면 해결되는 경우가 있는데, 이럴땐 let이 정말 필요한 상황인지 꼭 점검해야한다. **모던 웹개발에서 초보자가 let을 써야만된다면, 대부분의 경우엔 설계를 잘못된 것이다.**

그럼 잘 쓰지도 않는 let은 대체 왜 이제껏 설명한 것인가? 사실 필자는 훨씬 중요한 const를 먼저 설명해보려고 노력했으나, let을 미리 설명하지 않고는 도저히 const를 설명하기가 힘들었을 뿐이다.

코드로 let과의 차이점을 확인해보자. let을 사용했던 코드를 const로 바꿔보았다.

```
1  const a, b;
2  a = 1;
3  a = 2;
4  b = 3;
5  console.log(a, b, a + b);
```

빨간색 물결 밑줄이 있다는 건, 에러가 있다는 뜻이다. 무슨 에러인지 확인하기 위해, 오른쪽 상단 Errors 탭을 클릭해보자.



개발자에게 에러는 적이자 친구이다. 단언컨데, **우리의 개발 실력은 에러를 얼마나 많이 해결해 봤는지(debug)에 따라 늘 것이다.** 영어라서 겁이 나는가? 미안하지만, 개발자는 영어와 친숙해져야된다. 그렇다고 영어를 매우 잘하라는 뜻이 아니다. 나도 구글번역과 네이버 사전의 도움을 많이 받는다.

여기서 type 에러는 지금 공부할 단계가 아니다. 중요한 에러는 크게 두가지로 볼 수 있다.

1. 'const' declarations must be initialized.

const 선언은 반드시 초기화되어야 합니다.

line2에서 분명 초기화하지 않았는가? 그러나, const는 그런 거 허용하지 않는다. 반드시, 선언부터 초기화까지 한 줄에 마쳐야한다. **const는 이처럼, 여러분이 초기화하지 않고 변수를 사용하는 실수를 막아준다.**

2. Cannot assign to 'a' because it is a constant.

a에 assign할 수 없습니다. 왜냐면, constant 이기 때문입니다.

assign은 디버깅하다가 많이 보게 될 단어인데, 한국어로 '배정'이다. 쉽게 말하면 '무언가를 집어넣는다'는 뜻이라고 이해하면 된다. constant는 const로 선언된 변수를

의미한다. 풀어서 이해해보면,

a에 집어넣을 수 없습니다. 왜냐면, constant이기 때문입니다.

이미 1을 집어넣었기 때문에, 2를 집어넣으려 해도 안 들어가는 것이다.

해당 에러를 수정해 코드를 다시 써보면 다음과 같다.

```
1  const a = 1, b = 3;  
2  console.log(a, b, a + b);
```

결과:

```
[LOG]: 1, 3, 4
```

즉, const는 let과는 다른 두가지 특징이 있는 것으로 보이는데,

1. 선언과 동시에 초기화를 마쳐야한다.
2. 한번 값을 집어넣었으면 바꿀 수 없다.

사실, 2는 틀렸다. 나중에 배열(array), 객체(object)를 배우게 되면 const로 선언된 변수를 바꿀 수 있다. 그리고 배열과 객체의 형태를 const로 충분히 바꿀 수 있기 때문에, 90%의 경우엔 let을 쓸 필요 없이 const만으로 충분하다. 여러분은 배열과 객체를 배우지 않았기에 지금 설명하긴 너무 어려우니 일단은 넘어가자.

\* const는 constant 의 줄임말이다. 즉, 특수한 변수인 '상수'인데, 아마 기초 웹개발하면서 여러분은 써볼 일이 거의 없을 경우인, 걱정하고 상수로 쓸 경우를 제외하면 개발자들은 const로 선언된 것을 '변수'라고 부른다.

## 2-3. var

가끔 다른 개발자들이 쓴 옛날 코드를 볼 때가 있는데, let도, const도 아닌 var를 사용해 변수 선언하는것을 볼 수 있다. var는 2014년까지 쓰이던 구식 변수 선언법이고, 언어론적으로 자세하게 설명하진 않겠지만, 매우 불안정하고 위험하니 **쓰지마라**. 단, 옛날 코드를 볼 때 var가 나왔다면, 변수를 선언했구나 하고 이해하면 된다.

이제 변수에 숫자 말고 다른 것들을 집어넣어보자. 다음 주제인 자료형(Data Type)에서 다룰 것이다.

핵심정리:

- const - 대부분의 변수를 선언할 때 사용
- let - 웹개발에서 let을 써야만되는 경우라면, 대부분의 경우,  
설계를 잘못 한 것이다.
- var - 사용하지 말 것

개발 실력은 디버그를 많이 할수록 는다.



### 3. 자료형(Data Type)과 백틱(` `), 비교연산자

우리가 선언한 변수엔 숫자 말고도 다양한 '자료형(Data Type)'을 넣을 수 있다. 여기서, Type이라는 단어가 처음 등장한다는것에 주목하자. 우리가 배우는 것은 TypeScript 라는 언어다. TypeScript 개발팀이 Type을 얼마나 중요하게 생각했는지, 언어 이름에 Type을 쓸 정도였다. 글을 읽다보면, 자료형을 '타입'이라고 부를 일도 있을 것이다.

쓸 수 있는 자료형은 다음과 같다. 사실 더 많지만, 지금 당장 알아둬야 될 것들만 썼다.

number - 숫자를 의미.

0, 1, 2, -1, -2.58, ...

string - '실'이라는 뜻인데, '문자열'을 의미한다.

"a", "hello", 'TypeScript', '0'

큰따옴표로 쓰던, 작은따옴표로 쓰던 똑같이 string이다. 관습적으로, **꼭 필요한 경우가 아니라면 작은따옴표('')만 쓴다.**

**따옴표를 생략하면 키워드나 변수로 인식하므로, 변수를 string으로 쓰고싶으면 반드시 따옴표를 써야 한다.**

**중요! 0을 따옴표로 감싸면 number가 아닌 string으로 인식한다.**

boolean - true 또는 false를 말한다. 참과 거짓.

**따옴표를 써서 'true'라고 쓰면 boolean이 아닌 string으로 인식한다.**

array, object, function - 역시 자료형이나, 난이도가 있는 주제이므로 각각의 장에서 따로 다룬다.

\* 키워드(keyword)는 특정한 작업을 하는 단어들을 말한다. 우리가 지금까지 배운 키워드는 const, let, var이며, 앞으로 더 많은 키워드들을 배워볼 것이다.

typeof 를 통해 내가 쓴 자료형이 무엇인지 확인해보자.

실습1)

우리에겐 둘 다 0일 뿐인데, 컴퓨터는 number 0과 string '0'을 다르게 인식한다.

```
1 const a = 0;
2 console.log(typeof a);
```

결과:

```
[LOG]: "number"
```

```
1 const a = '0';
2 console.log(typeof a);
```

결과:

```
[LOG]: "string"
```

둘을 비교해서 같은지 알아보자. ===를 썼는데, 이건 다음 장에서 배울 비교연산자라고 하는 것이다. 결과가 맞으면 true, 틀리면 false다.

```
1 console.log(0 === '0');
```

결과:

```
[LOG]: false
```

실습2)

string 은 항상 따옴표를 써줘야한다.

다음은 에러다.

```
1 const a = david;  
2 console.log(typeof a);
```

```
Cannot find name 'david'.
```

'david'를 찾을 수 없습니다.

david는 키워드도 아니고, 변수도 아니다. 우린 david를 선언한 적이 없다.

```
1 const a = 'david';  
2 console.log(typeof a);
```

결과:

```
[LOG]: "string"
```

실습3)

boolean도 확인해보자.

```
1 const a = true;  
2 console.log(typeof a);
```

결과:

```
[LOG]: "boolean"
```

하지만 다음과 같이 따옴표를 써서 'true' 라고 쓰면 string이다.

```
1 const a = 'true';
2 console.log(typeof a);
```

결과:

```
[LOG]: "string"
```

백틱(``)에 대해 알아보자. 주로 문자열 안에서 변수나 다른 것을 좀 더 편하게 사용하려고 쓴다. 백틱은 탭키 바로 위에, 작은따옴표처럼 생겼다.

백틱을 쓰지 않고 문자열을 여러개 이어붙이려면 '+' 를 사용해, 여러개 문자열을 이어붙여야 한다.

```
1 const myName = '조교행님';
2 const age = 28;
3
4 console.log(myName + '은 ' + age + '살이다.');
```

결과:

```
[LOG]: "조교행님은 28살이다."
```

백틱을 사용하면, 문자열을 여러개 나누지 않고, 한번에 끝낼 수 있다.

```
1 const myName = '조교행님';
2 const age = 28;
3
4 console.log(`${myName}은 ${age}살이다.');
```

결과:

```
[LOG]: "조교행님은 28살이다."
```

백틱 안에서 변수를 표현할 땐 이와 같이, \${변수이름}을 사용한다.

또한 백틱을 사용하면 백틱 안에서 엔터를 쳤을 때 줄바꿈이 그대로 적용되므로, console.log()를 몇 줄씩이나 쓸 필요가 없다.

```
1 const myName = '조교행님';
2 const age = 28;
3
4 console.log(`
5     ${myName}은
6     ${age}살이다.
7 `);
```

결과:

```
[LOG]: "  
    조교행님은  
    28살이다.  
"
```

엔터나 탭이 그대로 적용되었다.

핵심정리:

string은 작은따옴표를 쓴다. 따옴표가 없으면 string이 아니다.

백틱을 사용하면, 문자열을 여러개 나누지 않고, 한번에 끝낼 수 있다.

```
'0' === 0    // false
```

## 4. if, else, ||, &&, ?, :

저번 시간에 잠깐 등장한 비교연산자를 알아보자. 주로 쓰게 될 비교연산자는 다음과 같다.

```
a === b      // = 3개 사용.           같으면 true, 다르면 false
a !== b      // ! 1개, = 두개 사용.   다르면 true, 같으면 false
a > b        // a가 b보다              크면 true, 작으면 false
a >= b       // a가 b보다 '같거나'     크면 true, 작으면 false
```

\* 3개의 === 가 쓰여져있다는건, = 와 ==(2개) 와 ===(3개) 의 의미가 다르다는 것을 의미한다.

= 1개는 우리가 이미 배웠듯이, '집어넣는다' 는 의미이므로, 완전히 다른 의미이다.

== (2개)는 값 비교이다. 0 == '0' 은 type이 다르지만 true이다.

=== (3개)는 type까지 비교한다. 0 === '0' 은 type이 다르기 때문에 false다.

대부분의 경우엔, ==(2개)를 쓰지 않고, ===(3개)만 쓴다.

이제 비교연산자를 사용해 조건문(if, else)을 연습해보자.

조건문을 사용하는 대표적인 예는 로그인이다.

로그인 시 아이디와 비밀번호가 일치하면(if)

DB에서 사용자의 모든 정보를 가져온 다음, 메인 화면으로 이동

로그인에 실패하면(else)

에러메세지

조건문은 다음과 같이 생겼다.

```
1  if (condition) {
2      block;
3  } else {
4      block;
5  }
```

condition은 조건이다. 조건문은 조건(condition)이 있어서 조건문이다.

조건이 맞으면 if문을 실행시키고,

조건이 틀리면 else를 실행시킨다.

코딩 습관 들이기:

line1:

if 의 f뒤엔 한 칸 띄우고 소괄호를 쓴다.

소괄호에서 중괄호가 이어지는 부분도 한 칸 띄어준다.

if else 다음에 한 줄이 오면 중괄호를 생략할 수 있다.

하지만, 코드를 읽기 쉽게 하기 위해 항상 중괄호를 쓰는 건 좋은 습관이다.  
line3: else는 if문의 닫는중괄호 바로 뒤에 써준다. else 앞뒤로 한 칸씩 띄어준다.  
else는 if가 끝나는 중괄호 바로 뒤에 둔다.

코드1) 숫자 연습

```
1  const age = 28;
2
3  if (age > 20) {
4      console.log('술 판매 가능');
5  } else {
6      console.log('영업정지 뽀뽀뽀');
7  }
```

결과:

```
[LOG]: "술 판매 가능"
```

우리가 설정한 조건은, 20보다 커야한다는 것이다. 따라서 if문이 실행되었다.

미성년자 나이로 바꿔보자.

```
1  const age = 17;
2
3  if (age > 20) {
4      console.log('술 판매 가능');
5  } else {
6      console.log('영업정지 뽀뽀뽀');
7  }
```

결과:

```
[LOG]: "영업정지 뽀뽀뽀"
```

조건에 맞지 않는 나이이기 때문에, else문이 실행되었다.

그러나, 20을 넣을 경우, 조건에 알맞지만 else문이 실행되는 문제가 있으므로, 20도 가능하도록 조건을 고쳐보겠다.

```

1  const age = 20;
2
3  if (age >= 20) {
4      console.log('술 판매 가능');
5  } else {
6      console.log('영업정지 뽀뽀뽀');
7  }

```

결과:

```
[LOG]: "술 판매 가능"
```

부등호(>, <) 오른쪽에 등호(=) 를 붙여주면, 딱 그 숫자일때도 포함된다.

숫자뿐만 아니라, 문자열로 조건문을 써보자. 이번엔 부등호 말고 (===)를 사용할 것이다.

코드2) 문자열 연습

```

1  const myName = '조교행님';
2  const age = 28;
3
4  if (myName === '조교행님') {
5      console.log('조교행님이 맞군. ');
6  } else {
7      console.log('누구나 넌');
8  }

```

결과:

```
[LOG]: "조교행님이 맞군. "
```

이 경우엔, myName이 '조교행님'이기때문에, if 문이 실행되었다.

\* 초보자가 흔히 하는 실수중에, 조건부에서 =를 하나만 쓰는 경우가 있다. 이 경우, 비교가 아니라 '대입'이 되어버린다.

myName을 '김선생'으로 바꿔보겠다.

```

1  const myName = '김선생';
2  const age = 28;
3
4  if (myName === '조교행님') {
5      console.log('조교행님이 맞군. ');
6  } else {
7      console.log('누구나 넌');
8  }

```

결과:

```
[LOG]: "누구나 넌."
```

밑줄 쳐진 예러는 무시하자. 이 경우엔, myName이 '조교행님'이 아니므로, else문이 실행된다.

코드3) !== 연습

(===)의 반대다. 해당 조건이 false일 경우 실행된다.

```
1  const myName = '김선생';
2  const age = 28;
3
4  if (myName !== '조교행님') {
5    console.log('조교행님이 아니군. ');
6  } else {
7    console.log('조교행님이군. ');
8  }
```

결과:

```
[LOG]: "조교행님이 아니군."
```

코드4) 비교연산자 없는 조건문

이제 좀 흥미로운 경우를 살펴보겠다.

```
1  const myName = '조교행님';
2
3  if (myName) {
4    console.log('이름이 있군요. ');
5  } else {
6    console.log('이름이 뭐예요? ');
7  }
```

결과:

```
[LOG]: "이름이 있군요."
```

비교연산자는 없다. myName 이라는 조건이 true이기 때문에 if문이 실행되었다. 무슨 의미일까?

여기선, 뭐라도 들어있으면 true로 받아들인다.

다음 경우를 보자.



```

1  const myName = '';
2
3  if (myName) {
4      console.log('이름이 있군요.');
```

결과:

```
[LOG]: "이름이 뭐예요?"
```

myName은 아무것도 없으므로 조건의 결과는 false 이다. 그래서 else문이 실행된다.

\* 조건문에서 false로 인식하는 것들이 몇가지 있는데, 다음은 꼭 알아두자.

false	(boolean)
0	(number)
''	(string. 빈 문자열)
undefined, null	(다음 장에서 배울 것.)

코드5) 여러개의 조건

여러가지 조건을 연달아 쓸 땐, else if 를 사용한다.

```

1  const myName = '조교행님';
2
3  if (myName === '조교행님') {
4      console.log('조교행님이군.');
```

myName 을 바꿔보며 결과를 비교해보자.

코드6) 조건 안에 조건

if 안에 if를 쓸 수 있다.

```

1  const myName = '조교행님';
2  const day = '일요일';
3
4  if (myName === '조교행님') {
5      if (day === '일요일') {
6          console.log('오늘은 출근 안함...');
7      } else {
8          console.log('출근이다. 으악!');
9      }
10 } else {
11     console.log('누구나 넌. ');
12 }

```

결과:

```
[LOG]: "오늘은 출근 안함..."
```

myName 이 '조교행님'이면, 그 안에 if문에서 day 가 '일요일'인지 아닌지 확인한다. day와 myName을 바꿔가면서 테스트해보자.

||, &&

여러가지 조건을 결합할 수 있다.

첫번째식 || 두번째식      첫번째식이 true이거나, 두번째식이 true 면 실행된다. (or)

첫번째식 && 두번째식      첫번째식이 true이고, 두번째식이 true 면 실행된다. (and)

무슨말인지 잘 이해가 안되니 코드를 통해 확인해보자.

코드7)

```

1  const myName = '조교행님';
2  const yourName = '선생';
3
4  if (myName === '조교행님' || yourName === '학생') {
5      console.log('|| 실행');
6  } else {
7      console.log('error!');
8  }

```

결과:

```
[LOG]: "|| 실행"
```

myName은 '조교행님'이 맞는데, yourName은 '선생'이 아니다. 그래도 실행된다. 조건이 true이기 때문이다.

||를 &&로 바꿔보자.

```

1  const myName = '조교행님';
2  const yourName = '선생';
3
4  if (myName === '조교행님' && yourName === '학생') {
5      console.log('&& 실행');
6  } else {
7      console.log('error!');
8  }

```

결과:

```
[LOG]: "error!"
```

&&는 양쪽 다 true여야 한다. myName은 '조교행님'이 맞는데, yourName은 '선생'이 아니기때문에 if문이 실행되는게 아니라 else 문이 실행되었다.

이제 조건문을 짧게 줄인 형태인 삼항연산자 ? : 를 써보겠다. 뭔지 설명하기 전에 코드부터 보자. 코드7번을 삼항연산자로 고쳐보면 다음과 같다.

```

1  const myName = '조교행님';
2  const yourName = '선생';
3
4  // if(myName === '조교행님' || yourName === '학생') {
5  //     console.log('||실행');
6  // } else {
7  //     console.log('error!');
8  // }
9
10 myName === '조교행님' || yourName === '학생' ? console.log('||실행') : console.log('error!');

```

line4 - 8      드래그해서 ctrl + / 하면 주석처리되며, 코드는 실행되지 않는다.

결과:

```
[LOG]: "||실행"
```

분석해보자.

```

myName === '조교행님' || yourName === '학생' ? console.log('||실행') : console.log('error!');
      조건                                true일 경우                        false일 경우

```

true일 경우,      ? 다음이 실행된다.

false일 경우,    : 다음이 실행된다.

와, 정말 좋은 걸 배웠으니 if else 를 버리고 삼항연산자 ? : 만 사용하면 될까? 그렇지 않다. 코딩하다보면 매우 짧게 조건문을 쓸 일이 있는데, 그 때는 삼항연산자를 사용하고, 조건이 긴 경우엔 if else 를 사용하면 된다.

이제 우리는 단순히 Data가 아닌, 여러개 데이터를 모아놓은 자료형인 배열(array)과 객체(object)를 알아볼 것이다. 먼저 배열부터 살펴보자.

핵심정리:

==(2개)를 쓰지 않고, ===(3개)만 쓴다.

조건이 맞으면 if문을 실행시키고,

조건이 틀리면 else를 실행시킨다.

조건문에서 false로 인식하는 것

false (boolean)

0 (number)

'' (string. 빈 문자열)

undefined, null (다음 장에서 배울 것.)

삼항연산자 ? : 는 조건문을 짧게 줄인 것이다.

## 5. 배열(array)

요일을 저장해보자. 요일은 총 7개다.

지금까지 배워왔던 방식으로 다음과 같이 저장할 것이다.

```
1  const mon = '월';
2  const tue = '화';
3  const wed = '수';
4  const thu = '목';
5  const fri = '금';
6  const sat = '토';
7  const sun = '일';
8
9  console.log(mon, tue, wed, thu, fri, sat, sun);
```

결과:

```
[LOG]: "월", "화", "수", "목",
      "금", "토", "일"
```

하지만, 이렇게 하나하나 적는것은 비효율적이다. 대신, 다음과 같이 한번에 저장할 수 있다.

```
1  const daysOfWeek = ['월', '화', '수', '목', '금', '토', '일'];
2
3  console.log(daysOfWeek);
```

결과:

```
[LOG]: ["월", "화", "수", "목",
      "금", "토", "일"]
```

이걸 배열(array)이라고 한다. **여러개의 data를 대괄호 [ ] 를 사용해 하나로 묶으면 배열이다.**

코딩 습관 들이기:

변수 이름은 길게 지어도 좋으니, 한눈에 알아볼 수 있도록 짓고, 자동완성을 적극 활용해 에러를 줄인다.

ex)   dow                   (x) - 월 의미하는지 모름.  
      daysOfWeek       (o)

변수 이름을 지을 때, 여러개 단어로 이어서 지을 경우

맨 첫글자는 소문자

띄어쓰기 하지 말고 이어서 쓸 것

이어지는 단어의 첫글자는 대문자 사용

이렇게 이름지으면 마치 낙타같이 보여서, 개발자들은 낙타법(camelCase) 라고 부른다.



ex)    days of week    (x)  
      days Of Week    (x)  
      daysofweek       (x)  
      days-of-week    (x)  
      days\_of\_week    (x)  
      DaysOfWeek       (x)  
      daysOfWeek      (o)

배열을 이루는 각각의 data를 요소(element)라고 한다. 요소에 접근하려면 **인덱스(index)**를 이용하면 된다. 배열 변수 옆에 대괄호를 쓴 숫자를 넣어준다.  
첫번째 요소인 '월'을 가져와보자.

```
1  const daysOfWeek = ['월', '화', '수', '목', '금', '토', '일'];  
2  
3  console.log(daysOfWeek[0]);
```

결과:

```
[LOG]: "월"
```

'월'의 인덱스는 0이다. **컴퓨터는 인간과는 다르게, 숫자를 셀 때 0부터 시작한다.** 그래서 인덱스 1은 '화'다.

만약, 없는 인덱스를 입력하면 다음과 같은 결과가 뜬다.

```
1  const daysOfWeek = ['월', '화', '수', '목', '금', '토', '일'];  
2  
3  console.log(daysOfWeek[7]);
```

결과:

```
[LOG]: undefined
```

**undefined는 정의되지 않았다는 뜻**이다. 우린 인덱스 7, 즉 인간 기준으로 8번째 데이터를 지정한 적이 없다.

\* 앞으로 디버깅하면서 **undefined** 를 지검도록 보게 될 것이니, 미리 이 단어를 알아두도록 하자.

\* undefined 과 비슷한 것으로 null 이라는 게 있다. 정의되지 않은 undefined과는 다르게, null은 개발자가 ‘일부러’ 비워둔 값이다.

배열의 요소는 const로 선언되었더라도 변경하거나 추가할 수 있다.

daysOfWeek 배열의 인덱스 0번을 변경해보겠다.

```
1  const daysOfWeek = ['월', '화', '수', '목', '금', '토', '일'];
2
3  daysOfWeek[0] = 'ㅋㅋㅋ';
4
5  console.log(daysOfWeek);
```

결과:

```
[LOG]: ["ㅋㅋㅋ", "화", "수",
"목", "금", "토", "일"]
```

배열에 데이터를 추가해보겠다. push() 를 사용한다.

```
1  const daysOfWeek = ['월', '화', '수', '목', '금', '토', '일'];
2
3  daysOfWeek.push('새로운요일!');
4
5  console.log(daysOfWeek);
```

결과:

```
[LOG]: ["월", "화", "수", "목",
"금", "토", "일", "새로운요일!"]
```

\* push를 사용하는 도중, 마침표( . )를 사용했다. 마침표는 객체에 접근할 때 사용되는것이다. 잘 생각해보면, 아무 생각없이 쓰고 있던 console.log() 에도 마침표가 있었다. 다음 장인 객체에서 좀 더 살펴보기로 한다.

배열의 요소는 바꿀 수 있어도, 배열 자체를 바꿀 수는 없다.

```
1  const daysOfWeek = ['월', '화', '수', '목', '금', '토', '일'];
2
3  daysOfWeek = ['크리스마스', '석가탄신일', '내 생일'];
```

다음과 같은 에러가 발생한다.

## Errors in code

Cannot assign to 'daysOfWeek' because it is a constant.

const 로 선언되었기 때문이다. 배열의 요소는 바꿀 수 있지만, 배열로 선언된 변수에 다른 걸 집어넣을수는 없다.

문자열뿐만 아니라, 자료형은 어떤 것이든 다 가능하며, 하나의 배열에 같은 자료형일 필요는 없다. 우리가 만든 변수도 배열의 요소가 될 수 있으며, 배열 안에 배열이 있을수도 있다.

다음의 예를 보자.

```
1  const myNum = 2;
2
3  const justData = [
4      '짜장면',
5      myNum,
6      true,
7      '짬뽕',
8      -3.14,
9      [
10         '김치찌개',
11         '미역국',
12         '카레'
13     ],
14     '햄버거'
15 ];
```

코딩 습관 들이기:

다음과 같이, 하나의 작업을 여러 줄에 쓸 때, 칸을 띄워 보기 좋게 만드는 작업을 ‘들여쓰기’라고 한다. 보통 스페이스바 네칸을 띄워쓴다.

시작 괄호 바로 다음을 띄운다. line11, line13에서처럼, 마지막괄호는 따로 써준다.

같은 그룹에 있으면 같은 칸을 유지해야된다.

탭키를 사용해서 들여쓸수도 있는데, 탭이 스페이스 네칸으로 되는지 확인하고, 안되어있으면 반드시 설정해주어야한다.

line5: line1에서 선언한 변수가 배열의 요소가 되었다.

line5: true, false 역시 배열의 요소가 될 수 있다.

line9: 배열 안에 배열이 있을 수 있다. 이런 형태를 이차원배열이라고 한다.

그러나 저렇게 자료형을 뒤죽박죽 섞어서, 뭐가 뭔지 구분도 안되게 정신없이 쓸 일은 거의 없다. 계속 실습을 진행하기 위해 다음과 같이 일단 바꿔주자.



```

1  const justData = [
2      '짜장면',
3      '짬뽕',
4      [
5          '김치찌개',
6          '미역국',
7          '카레'
8      ],
9      '햄버거'
10 ];

```

만약, '김치찌개'에 접근하고 싶으면 어떻게 하면 될까? 인덱스를 두 번 써주면 된다.

```
12 console.log(justData[2][0]);
```

결과:

```
[LOG]: "김치찌개"
```

배열은 여러 개의 데이터를 하나로 묶어준다. 그러나 상황에 따라서, 배열보다 '객체(object)'를 사용하는게 더 효율적일 수 있다. 다음 장에서 살펴보자.

핵심정리:

여러개의 데이터를 대괄호로 묶으면 배열(array)이다.

배열의 접근은 인덱스(index)를 사용하며, 0부터 시작한다.

undefined는 정의되지 않은 것이고, null은 일부러 비운 것이다.

배열의 요소는 const로 선언했더라도 변경하거나 추가할 수 있다. 단, 배열 자체를 바꿀 순 없다.

변수 이름은 길게 지어도 좋으니, 한눈에 알아볼 수 있도록 짓고, 낙타법을 이용한다.

## 6. 객체(object)

JavaScript에서는, 다른 언어들, C, C++, C#, Java, Python에서 사용되는 개념들과 아주 크게 다른 개념이 두가지 있다.

1. 객체(object)
2. 함수(function)

그래서, 혹시 미리 다른 언어를 공부해본 사람들은 머릿속에 든 걸 잊어버리고 새로운 마음으로 들었으면 좋겠다.

또한 JavaScript, TypeScript 개발자는 객체와 함수를 ‘가지고 놀’ 정도로 잘 알아야 한다. 그래서 다른 장보다 다뤄야 할 개념도 많고, 난이도도 어렵다. 지금까지는 프로그래밍 언어의 기본을 알아보는 ‘순한맛’ 강의였다면, 지금부터는 ‘매운맛’이다. 이해 안되면 다시 읽어보고, 코드로 연습해보자.

객체(object)란 뭘까?

내 개인정보를 배열로 저장했다고 치자.

```
1  const myInfo = [  
2      '조교행님',  
3      28,  
4      true,  
5      [  
6          '아빠',  
7          '엄마',  
8          '멍멍이'  
9      ]  
10 ];
```

뭐가 뭘지 구분 가능한가? ‘조교행님’은 이름일것이고, 28은 나이, true는 뭘지 모르겠고, 배열은 가족으로 보인다. 이건 두가지 점에서 상당히 불편하다.

1. 데이터가 뭘 의미하는지 어렵짐작해야한다. line4는 대체 뭘 의미하는걸까?
2. 데이터에 접근할 일이 있을때는 인덱스를 일일이 기억하고 있어야한다.

ex) 이름에 접근                      myInfo[0]

객체(object): 키(key)와 값(value)으로 이루어진 프로퍼티(property) 모음. 중괄호 { } 로 프로퍼티들을 묶는다.

\* object 는 물건, 물체, 대상이라는 뜻이다. 한국어로 ‘객체’라는, 무슨뜻인지도 모를 단어로 번역되어 쓰여서 감이 잘 안 올수도 있다. ‘데이터에 키를 붙여놓은것들의 모음’

정도로 받아들이자.

정의만 읽어선 뭘 말인지 감이 오지 않는다. 직접 확인해보자.

```
1  const myInfo = {
2    name: '조교행님',
3    age: 28,
4    isGirlFriend: true,
5    familyMembers: [
6      '아빠',
7      '엄마',
8      '멍멍이'
9    ],
10   techStack: {
11     frontEnd: 'React',
12     backEnd: 'GraphQL',
13     dataBase: 'SQLite'
14   }
15 };
```

나에 대한 객체를 하나 만들었다. 아까와 비교해서 뭐가 달라진 것 같은가?

각각의 데이터에 ‘키’(key)가 있다. 28(value)은 age(key)를 뜻하는 것이었다. 아까 무얼 뜻하는지 몰랐던 true(value)는 사실, 여자친구의 유무를 뜻하는 isGirlFriend(key)였다. 이처럼, 이름을 달아두니깐 내 정보에 대해서 훨씬 이해하기 쉬워졌다.

객체에 대해 좀 더 자세히 알아보자.

배열은 대괄호[ ]를 사용하고, 객체는 중괄호{ }를 사용한다.

line2: 키는 영문으로만 쓰며, 따옴표( ‘ ’ )를 쓰지 않는다.

name: ‘조교행님’ // 이것은 key와 value로 이루어진 하나의 프로퍼티다.

여러개의 프로퍼티를 만들고 싶으면, 세미콜론( ; )이 아닌 콤마( , )로 구분한다.

line10: 어떤 자료형이든 value로 모두 허용된다. 객체 안에 또 객체를 쓸 수 있으며, 나중에 배열 함수도 가능하다.

\* property는 ‘속성’ 이라는 뜻이다.

\* 앞으로 코딩하다보면, 여러분이 상상하는 것 이상으로, 콤마( , ) 를 안 써줘서 생기는 예러가 많을 것이다.

데이터에 접근할때는 인덱스가 아닌, **마침표( . )**를 사용한다. 이것은 곧, **객체는 순서가 없다**는 걸 의미한다. 마침표 뒤에 프로퍼티의 키를 써주면 된다.

```
17 console.log(myInfo.name);
```

결과:

```
[LOG]: "조교행님"
```

당연히, 잘못된 키를 타이핑하면 undefined 가 나온다.

\* 그렇다면, console.log() 에서 console 역시도 하나의 객체였다는것을 알 수 있다.  
log()는 console 객체의 프로퍼티 중 하나이며, 다음 장에서 배울 '함수'이다.

객체의 프로퍼티는 const로 선언되었더라도 바꿀 수 있다.

```
1  const myInfo = {
2    name: '조교행님',
3    age: 28,
4    isGirlFriend: true,
5    familyMembers: [
6      '아빠',
7      '엄마',
8      '멍멍이'
9    ],
10   techStack: {
11     frontEnd: 'React',
12     backEnd: 'GraphQL',
13     dataBase: 'SQLite'
14   }
15 };
16
17 myInfo.techStack.frontEnd = 'Vue.js';
18
19 console.log(myInfo.techStack);
```

결과:

```
[LOG]: {
  "frontEnd": "Vue.js",
  "backEnd": "GraphQL",
  "dataBase": "SQLite"
}
```

line17: myInfo객체의, techStack 프로퍼티의, frontEnd 프로퍼티에 접근해서, 'React'를 'Vue.js' 로 바꾸었다.

그러나, 객체 자체를 바꾸진 못한다. 아래 코드를 통해 확인해보면,

```

1  const myInfo = {
2      name: '조교행님',
3      age: 28,
4      isGirlFriend: true,
5      familyMembers: [
6          '아빠',
7          '엄마',
8          '멍멍이'
9      ],
10     techStack: {
11         frontEnd: 'React',
12         backEnd: 'GraphQL',
13         dataBase: 'SQLite'
14     }
15 };
16
17 myInfo = {
18     name: 'Tom',
19     age: 22
20 };

```

다음과 같은 에러가 확인된다.

#### Errors in code

Cannot assign to 'myInfo' because it is a constant.

const로 선언되었기 때문이다. 객체의 프로퍼티는 바꿀 수 있지만, 객체로 선언된 변수에 다른 걸 집어넣을수는 없다.

그럼 객체가 항상 배열보다 최고인가? 꼭 그렇지만은 않다.

개발자는 상황에 따라, 배열과 객체 중 알맞은 자료형을 선택해야 한다. 이는 앞으로 실습을 진행하면서 알게 될 것이다.

다음장은, JavaScript의 꽃이라고 할 수 있는 함수(function)다.

핵심정리:

객체는 키(key)와 값(value)으로 이루어진 프로퍼티(property)의 집합이며, 중괄호 { } 로 프로퍼티들을 묶는다.

프로퍼티는 콤마( , )로 구분한다.

객체는 순서가 없다. 데이터에 접근할 때는 인덱스가 아닌, 마침표( . )를 사용한다.

객체의 프로퍼티는 const로 선언되었더라도 바꿀 수 있으나, 객체 자체를 바꾸진 못한다.

## tip. 객체와 배열에 대한 깊은 이해

이번 장을 마치기 전에, 배열과 객체에 대해 추가적으로, 알아두면 개발하다 쓸모있는 정보들을 모아놓았다.

### 1. length

.length 는 string 또는 array 의 길이다.

```
1  const menu = '알리오올리오스파게티';
2
3  console.log(menu.length);
```

결과:

```
[LOG]: 10
```

```
1  const menu = ['짜장', '짬뽕', '탕수육'];
2
3  console.log(menu.length);
```

결과:

```
[LOG]: 3
```

그러나, 객체의 길이를 구하진 못한다.

```
1  const info = {
2    name: '조교행님',
3    age: 28,
4    job: '대학조교'
5  };
6
7  console.log(info.length);
```

에러 내용:

#### Errors in code

```
Property 'length' does not exist on type '{ name: string; age: number; job: string; }'.
```

프로퍼티 length 는 존재하지 않습니다. / 타입 {~~} 에.

우리는 length 라는 프로퍼티 를 정의한 적이 없기 때문이다. 객체의 길이는 좀 다르게 구해야 된다. Object.keys(객체).length 를 이용할 것이다.

```

1  const info = {
2      name: '조교행님',
3      age: 28,
4      job: '대학조교'
5  };
6
7  console.log(Object.keys(info).length);

```

결과:

```
[LOG]: 3
```

length는 코딩하다 자주 쓰는 개념이니, 잘 알아두자.

2. 빈 배열 [ ] 과 빈 객체 { } 는 조건문에서 false 가 아니다.

```

1  const a = {};
2
3  if (a) {
4      console.log('run');
5  } else {
6      console.log('fail');
7  }

```

결과:

```
[LOG]: "run"
```

3. 동일해보이는 두 배열 또는 두 객체를 서로 비교할수는 없다.

```

1  const a = [1, 2, 3];
2  const b = [1, 2, 3];
3
4  if (a === b) {
5      console.log('same');
6  } else {
7      console.log('different');
8  }

```

결과:

```
[LOG]: "different"
```



```

1  const a = {
2    name: '조교행님',
3    age : 28
4  };
5  const b = {
6    name: '조교행님',
7    age : 28
8  };
9
10 if (a === b) {
11   console.log('same');
12 } else {
13   console.log('different');
14 }

```

결과:

```
[LOG]: "different"
```

이는 근본적으로, 두 변수의 주소가 다르기 때문이다. 주소에 대해 자세히 설명하면 지나치게 어려워진다. 확실한 건, 두 개의 객체가, 두 개의 배열이 서로 같은지를 비교하는 기능은 JavaScript에서 제공하지 않는다는 것이다.

4. 짧게 쓰기: 객체 프로퍼티의 키와 값이 동일한 이름일 경우

다음과 같은 경우가 있을 수 있다.

```

1  const id = 1;
2  const a = { id: id };
3  console.log(a);

```

결과:

```
[LOG]: {
  "id": 1
}
```

여기서, line2는 다음과 같이 줄일 수 있다.

```
2  const a = { id };

```

즉, { id: id } 는 { id } 와 같다.

응용해보자. 이미 존재하는 두 개의 변수를 하나의 객체로 만들려면 원래 이렇게 했어야 했다.

```

1  const myName = '조교행님';
2  const age = 28;
3  const member = {
4      myName: myName,
5      age: age
6  }
7
8  console.log(member);

```

이젠 다음과 같이 한다.

```

1  const myName = '조교행님';
2  const age = 28;
3  const member = { myName, age };
4
5  console.log(member);

```

결과:

```

[LOG]: {
  "myName": "조교행님",
  "age": 28
}

```

## 5. 구조 분해 할당

이미 존재하는 배열이나 객체를 새로운 배열이나 객체에 넣는 방법이다.

구조 분해 할당을 쓰지 않고, 다음과 같이 할 수 있다.

```

1  const data = [2, 5, 8, 52, 83, 96];
2
3  const x = data[0];
4  const y = data[1];
5  const z = data[2];
6  const other = [0, 0, 0];
7  other[0] = data[3];
8  other[1] = data[4];
9  other[2] = data[5];
10
11
12 console.log(x); // 2
13 console.log(y); // 5
14 console.log(z); // 8
15 console.log(other); // [52, 83, 96]

```

line6: 텅 빈 배열 [ ] 을 선언하면 안된다. 다음과 같이 초기화를 해줘야한다.

구조 분해 할당은 위 코드를 다음과 같이 간단하게 할 수 있다.

```
1  const data = [2, 5, 8, 52, 83, 96];
2  const [x, y, z, ...other] = data;
3  console.log(x, typeof x); // 2, number
4  console.log(y, typeof y); // 5, number
5  console.log(z, typeof z); // 8, number
6  console.log(other); // [52, 83, 96]
```

x, y, z엔 2, 5, 8이, other엔 [52, 83, 96] 이 들어갔다.

...변수이름 을 사용하면, 나머지 요소가 다 들어가서 배열을 이룬다.

x, y, z는 배열이 아니라 새로 만들어진 '변수'이며, 해당되는 자료형을 가진다.

객체도 가능하다.

```
1  const data = {
2    myName: '조교행님',
3    age: 28,
4    address: '경남 함양군',
5    isGirlFriend: true,
6    language: ['영어', '한국어', '중국어'],
7    skill: {
8      frontEnd: 'React',
9      backEnd: 'GraphQL'
10   }
11 };
12
13 const { address, age, myName, skill: { frontEnd }, ...other } = data;
14 console.log(myName, typeof myName);
15 console.log(age, typeof age);
16 console.log(address, typeof address);
17 console.log(frontEnd, typeof frontEnd);
18 console.log(other, typeof other);
```

결과:

```
[LOG]: "조교행님", "string"
[LOG]: 28, "number"
[LOG]: "경남 함양군", "string"
[LOG]: "React", "string"
[LOG]: {
  "isGirlFriend": true,
  "language": [
    "영어",
    "한국어",
    "중국어"
  ]
}, "object"
```

배열과는 다르게, 여기서 가져오고자 하는 프로퍼티 이름을 정확히 써줘야한다. 객체는 인덱스가 따로 없기 때문이다. 즉, 순서가 없으므로 프로퍼티 이름으로 접근해야한다.

그래서, 데이터를 가져올 때 자세히 보면 객체가 쓰여진 순서와 일치하지 않는다. data 객체는 myName, age, address 순인데, 가져올 땐 address, age, myName 으로 가져온다.

myName, age, address라는 새로운 '변수'를 만든다. 이것들은 더이상 '객체'가 아니라 '변수'이며, 각각의 자료형을 가진다.

객체 안에 객체의 프로퍼티에 접근해 새로운 변수를 만들려면, 위와 같이

객체이름: { 프로퍼티명 }

으로 접근해야한다.

남는것을 other라는 새로운 객체에 담는다.

## 6. optional chaining ?.

만약 있으면 접근하고, 없으면 undefined 반환

```
1  const info = {
2    name: '조교행님',
3    age: 28,
4    address: '경남 함양군',
5    skill: {
6      frontEnd: 'React',
7      backEnd: 'GraphQL'
8    }
9  };
10
11  const frontEnd = info?.skill?.frontEnd;
12
13  console.log(frontEnd);
```

info가 있으면(?) skill에 접근(.)하고, skill이 있으면(?) frontEnd에 접근하라(.)

이 기술은 주로 frontend 에서 undefined 를 화면에 붙일 때 생기는 에러를 막기 위해 쓰인다. 지금은 그 파워를 느끼기 어렵고, 이런 게 있다고만 알아두자.

## 7. 함수(function) 기본이론 1 - with TS

프로그래밍에서의 함수(function)는 중학교 수학시간 때 배운 함수와는 다르다. 둘 다 같은 영어단어를 써서 그게 그거 아닌가 생각할 수 있지만, 수학시간에 배웠던 함수를 프로그래밍에서의 함수에 끼워맞춰 설명하려다보면 오히려 더 복잡하고 어렵다.

프로그래밍에서 ‘함수’를 말하는 영어단어인 function은 ‘기능’으로 번역하는 게 오히려 더 정확하다. 차라리 수학적 함수와 구분되게 ‘평션’ 이라고 영어 그대로 부르는 게 더 낫다고 생각한다. 그러나, 국내에 존재하는 프로그래밍 서적이거나, 구글링해서 나오는 한국어 자료들은 전부 다 ‘함수’라고 부르는데, 나 혼자 평션이라고 부를 수는 없지 않은가. 아무튼, **수학적 함수와 프로그래밍의 함수는 다른 단어**라고 생각하자. 이제 함수에 대한 설명을 시작하겠다.

함수는 ‘기능’을 가진, ‘재사용’가능한 ‘도구’ 다. 코드로 확인해보자.

```
1 console.log('안녕, 조교행님. 나이는 28세');
2 console.log('안녕, 은지. 나이는 16세');
3 console.log('안녕, 현수. 나이는 21세');
4 console.log('안녕, 지원. 나이는 14세');
5 console.log('안녕, 준수. 나이는 23세');
```

여기서 공통된 건 무엇이 있을까?

1. 모두 console.log() 를 사용한다.
2. ‘안녕’ 이 앞에 들어가고, 그 뒤에 이름, 나이가 온다.

그런데, 내가 ‘안녕’ 대신 ‘하이’ 를 쓰고 싶다면, 또는 ‘나이’대신 ‘연세’를 쓰고 싶다면 나는 저 다섯개의 문장을 모두 ‘하이’와 ‘연세’로 바꿔줘야된다. 만약, 이러한 문장이 몇백개 되어서, 일일이 바꿔줘야 된다면 정말 귀찮을 것이다!

우린 아직 함수가 뭔지도 모르지만, 이름과 나이를 받아 와서 내가 무슨 이름과 나이를 주든 잘 출력되는 함수를 하나 작성해보겠다.

```
1 function sayHello(name: string, age: number) {
2   |   return `안녕, ${name}. 나이는 ${age}세`;
3   | }
4
5 const result = sayHello('조교행님', 28);
6
7 console.log(result);
```

결과:

```
[LOG]: "안녕, 조교행님. 나이는 28세"
```

이제 이 함수를 분석해보면서, 함수가 뭔지 알아보자. 어려울 수 있으니 집중해서 봐야한다.

```
1 function sayHello(name: string, age: number) {  
2     return `안녕, ${name}. 나이는 ${age}세`;  
3 }  
4  
5 const result = sayHello('조교행님', 28);  
6  
7 console.log(result);
```

노란색 박스가 두 개 있다. 첫번째 노란색 박스는 우리가 배우고자 하는 ‘함수’다. 아래에 있는 노란색 박스는 함수를 “호출(call)” 하는 코드이다. **함수를 쓰고싶으면 호출해야한다.**

호출하는 코드에 쓰이는 소괄호 ( ) 는 함수를 실행한다는 의미이다.

첫번째 노란색 박스는 function 이라는 키워드로 시작했다. 나중에 지금 방법보다 훨씬 많이 쓰는 방법인 ‘화살표 함수’ 를 만드는 법을 배우기 전까지는, 함수를 만들 땐 우선 function 을 맨 앞에 쓰고 시작한다고 알아두자. function 키워드로 함수 만드는 법을 잘 알아두면, 화살표 함수를 이해하는 데 크게 어렵지 않다.

function 다음에 나오는 건 함수의 이름이다. 이 함수의 이름은 sayHello 이다.

**함수 이름은 동사+명사 로 짓는다.** sayHello(), showMessage() ...

함수의 이름 뒤의 소괄호 ( ) 는 호출할 때 쓰이는 소괄호와 의미가 다르다. 이 안에 다음 이미지에서 설명할 매개변수를 담는다. 소괄호 뒤엔 중괄호 { } 로 함수의 내용이 들어간다.

```

1 function sayHello(name: string, age: number) {
2     return `안녕, ${name}. 나이는 ${age}세`;
3 }
4
5 const result = sayHello('조교행님', 28);
6
7 console.log(result);

```

매개변수(parameter)

인자(argument)

line5의 호출 코드에서 소괄호 ( ) 안에 콤마 , 로 구분되었던 '조교행님' 과 28 은 이제 **sayHello() 함수로 전달**될 것이다. 그 안에서, 다음과 같은 일이 일어난다.

'조교행님'	name 이라는	string 형 변수가 된다.
28	age 라는	number 형 변수가 된다.

line5 '주는 쪽'인 함수 호출에 쓰인 '조교행님' 과 28은 인자(argument, 실매개변수) 다.  
 line1 '받는 쪽'인 sayHello() 함수의 소괄호 ( ) 안에 쓰인 name 과 age는 매개변수(parameter, 형식매개변수) 다.

인자(argument)	: 주는 쪽. '조교행님', 28
매개변수(parameter)	: 받는 쪽. name, age

엄청 어려운 단어들이 나온다고 스트레스받을 필요는 없다. 개발자들도 parameter와 argument의 차이를 잊어버려서 서로 섞어서 쓰기도 한다. 이 둘을 각각 형식매개변수, 실매개변수라고 부르기도 하므로, 더 헷갈린다. 용어보다 중요한 건, **호출 시에 소괄호 ( ) 안에 있는 값이 함수의 소괄호 ( ) 속으로 '전달' 되었다는 것이다.**

최초로, JavaScript 가 아닌 TypeScript 문법이 등장했다. 바로, parameter 옆에 있는 타입(string, number)이다. TypeScript에서는, JavaScript와는 다르게 업그레이드 된 기능이 있다.

1. parameter의 형태를 지정해줘야 한다.  
 name: string 이라는 뜻은, name에 다른 타입은 받지 않고 오직 string 만 받겠다는 뜻이다.
2. argument 갯수를 지켜줘야한다. sayHello() 는 정확히 2개의 argument 를 원한다. 1개나 3개를 넘겨주면 에러가 발생한다.



```

1 function sayHello(name: string, age: number) {
2     return `안녕, ${name}. 나이는 ${age}세`;
3 }
4
5 const result = sayHello('조교행님', 28);
6
7 console.log(result);

```

A diagram with two purple arrows. One arrow starts from the `name` parameter in the function signature on line 1 and points down to the `${name}` placeholder in the template literal on line 2. The other arrow starts from the `age` parameter in the function signature on line 1 and points down to the `${age}` placeholder in the template literal on line 2.

`name`과 `age`는 함수 `sayHello()` 의 변수이기 때문에, `sayHello()` 안에서 자유롭게 쓸 수 있다.

```

1 function sayHello(name: string, age: number) {
2     return `안녕, ${name}. 나이는 ${age}세`;
3 }
4
5 const result = sayHello('조교행님', 28);
6
7 console.log(result);

```

A diagram with a black arrow. It starts from the return value of the `sayHello` function call on line 5 and points down to the `result` variable in the assignment `const result =`.

`return` '반환' 이라고 번역된다. `sayHello()` 함수가 '내뱉는' 결과다.

line5: `sayHello()`의 결과를 `result` 라는 변수로 입력받았다. `result`의 타입은 무엇일까? `string` 이다.

최종적인 결과:

```
[LOG]: "안녕, 조교행님. 나이는 28세"
```



처음으로 돌아가서, 결과만 생각해보자.

# sayHello('조교행님', 28)

sayHello() 함수 괄호 안에 두 개의 인자, '조교행님', 28 을 집어넣어 나온 return 값은?

“안녕, 조교행님. 나이는 28세”

함수 장을 맨 처음 시작할 때, 함수는 뭐라고 정의했는가?

함수는 ‘기능’을 가진, ‘재사용’가능한 ‘도구’

그러니까, sayHello() 함수에 이름과 나이를 집어넣었더니, 이름과 나이가 들어간 string 타입의 “문장”이 나왔다. sayHello() 는 이런 ‘기능’을 가진 함수다.

그럼 기능을 가진 건 알겠고, 재사용은 어떻게 하는걸까? 직접 보자.

```
1 function sayHello(name: string, age: number) {  
2   return `안녕, ${name}. 나이는 ${age}세`;  
3 }  
4  
5 const a = sayHello('조교행님', 28);  
6 const b = sayHello('은지', 16);  
7 const c = sayHello('현수', 21);  
8 const d = sayHello('지원', 14);  
9 const e = sayHello('준수', 23);  
10  
11 console.log(`  
12   ${a}  
13   ${b}  
14   ${c}  
15   ${d}  
16   ${e}  
17 `);
```

결과:

```
[LOG]: "  
  안녕, 조교행님. 나이는 28세  
  안녕, 은지. 나이는 16세  
  안녕, 현수. 나이는 21세  
  안녕, 지원. 나이는 14세  
  안녕, 준수. 나이는 23세  
"
```

함수를 여러번 만들었는가? 아니다. **딱 한번만** 만들었다. 변수 a - e 까지 만들어서 **“재사용”**했다.

이처럼, **함수는 ‘기능’을 가진, ‘재사용’가능한 ‘도구’**다.

이제 맨 처음의 문제를 해결해보자. ‘안녕’을 ‘하이’로, ‘나이’를 ‘연세’로 바꾸려면, 해당 함수만 바꿔주면 된다. 함수 자체를 바꾼것이다.

```
1 function sayHello(name: string, age: number) {  
2   |   return `하이, ${name}. 연세는 ${age}세`;  
3 }  
4  
5 const a = sayHello('조교행님', 28);  
6 const b = sayHello('은지', 16);  
7 const c = sayHello('현수', 21);  
8 const d = sayHello('지원', 14);  
9 const e = sayHello('준수', 23);  
10  
11 console.log(`  
12   |   ${a}  
13   |   ${b}  
14   |   ${c}  
15   |   ${d}  
16   |   ${e}  
17   | `);
```

결과:

```
[LOG]: "  
  하이, 조교행님. 연세는 28세  
  하이, 은지. 연세는 16세  
  하이, 현수. 연세는 21세  
  하이, 지원. 연세는 14세  
  하이, 준수. 연세는 23세  
"
```

다섯개의 문장이든, 백개의 문장이든, 일일이 다 바꿔줄 필요 없이 함수만 바꿔주면 된다.

\* 개발자가 아닌 사용자가, sayHello() 가 어떤 코드로 작성되었는지 꼭 알아야 사용할 수 있을까? 절대 아니다! 사용자는, sayHello() 를 어떻게 쓰는지 사용법은 알아야하지만, sayHello() 의 함수 코드를 알 필요가 없다. 우리 역시도 마찬가지였다. console.log() 에서 log() 역시 함수다. 정확하게는, 객체 console의 프로퍼티 중 하나인 함수다. 뭔가를 넣으면 그걸 출력해준다는걸 우린 알고있고, 사용법도 알고있지만, log() 가 어떻게 생겼는지는 우리도 모른다.

궁금하니깐 한번 찍어보자.

```
console.log(console.log);
```

 // 참고로, log 뒤에 소괄호 () 를 안 붙였는데, 붙이면 함수가 실행되어버리기 때문이다.

결과:

```
[LOG]: function (...objs) {
    const output = produceOutput(objs);
    const eleLog = eleLocator();
    const prefix = `[$${id}]: `;
    const eleContainerLog = eleOverflowLocator();
    allLogs.push(`${prefix}${output}`);

    eleLog.innerHTML = allLogs.join("
");

    if (autoScroll && eleContainerLog) {
        eleContainerLog.scrollTop = eleContainerLog.scrollHeight;
    }
    raw[name](...objs);
}
```

뭔소린지 알겠는가? 나도 모른다! log() 는 이렇게 생긴 놈이지만, 우린 모르는데도 잘만 쓰고 있었다. 마치 스마트폰의 회로 원리는 모르지만 카카오톡은 잘만 쓰는것과 같다. 함수 사용자는 사용법만 알고 있으면 된다.

이제 함수가 무슨 놈인지는 알게 되었으니, 함수에 대해 꼭 알아야 할 내용을 좀 더 자세히 알아보자.

핵심정리:

수학적 함수와 프로그래밍의 함수는 다른 단어

프로그래밍의 함수: '기능'을 가진, '재사용'가능한 '도구'

함수는 미리 만들어놨다가 필요할 때 호출해서 사용

함수 이름은 동사+명사 로 짓는다

사용자는 인자(argument)를 입력해 함수로 보냄

함수 사용자는 함수 사용법만 알고 있으면 되며, 내부 코드는 알 필요가 없음

TypeScript에선, 함수의 parameter 타입을 지정해야함

## 8. 함수(function) 기본이론 2 - with TS

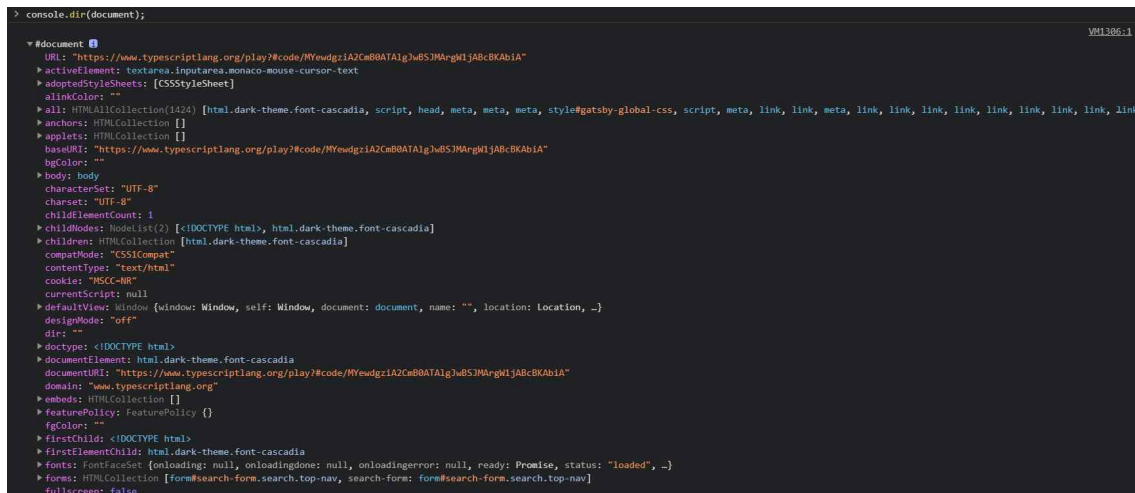
이 장에서는 지난 장에서 설명 안했지만, 함수에 대해 꼭 알아야하는 기본개념에 대해 다룬다.

## 1. 내장객체(Built-in object) 와 DOM

지난 장의 마지막은 `console.log()` 에서 `log()` 의 정체에 대해 알아보는 것이었다. 그런데 잘 생각해보면, 우린 `console` 객체를 만든적도, `log()` 함수를 만든적도 없는데 잘만 쓰고 있었다. 왜냐면 `log()` 는 JavaScript에서 기본적으로 제공하는 `console` 내장객체(**Built-in object**)의 함수이기 때문이다. 아파트 전월세 구하러 다니다보면 빌트인 가전이라고 해서 미리 갖추어진 냉장고, 에어컨, 세탁기 등을 볼 수 있는데, 이런 것들은 건설회사에서 제공하는 것이기 때문에 이사 들어올 때 따로 사야 할 필요없이 그냥 쓰면 된다.

JavaScript에도 우리의 편의를 위해 이미 제공되는 내장객체들이 있다.

대표적으로 웹페이지를 조작할 때 쓰는 document 객체가 있다. document 객체는 현재 사이트의 정보를 보여준다. 클릭, 색깔 바꾸기, 스크롤, 화면 사이즈 조작, 입력 등 웹페이지에 관련된 작업을 할 때, 순수한 JavaScript, TypeScript로만 개발하게 되면 보통 document.querySelector() 로 해당 HTML class에 접근해서 작업하게 되는데, 이를 DOM 조작이라고 한다.



크롬 개발자 도구에서 `console.dir(document)` 를 통해 찍어본, 현재 우리가 사용하는 TypeScript Playground 웹페이지의 `document` 객체의 일부. 이러한 정보들이 모여서 우리가 사용하는 TypeScript Playground를 이룬다.

원래 JavaScript는 탄생 목적 자체가 DOM 조작을 하기 위함이다. 그러나 우리의 코스에선 다른 책에서 필수적으로 다루는, `querySelector()` 를 사용해 직접 DOM 에 접근해 조작하는 내용을 다루지 않을 것이다. 우리가 사용할 예정인 React에선 다른 방식을 사용하기 때문이다.

## 2. 지역변수(local variable) vs 전역변수(global variable)

이론적으로 어렵게 다루면 꽤 어려워지는 부분이다. 꼭 필요한 것만 알고 가자.

우선 다음 코드를 보자.

```
1  const a = 1;
2  const b = 2;
3
4  function showResult(a: number, b: number) {
5      a = 3;
6      b = 4;
7      return [a, b];
8  }
9
10 const [resultA, resultB] = showResult(a, b);
11
12 console.log(`
13     a: ${resultA}
14     b: ${resultB}
15 `);
16
17 console.log(`
18     a: ${a}
19     b: ${b}
20 `);
```

line10: 여러개의 return 값을 받을 땐, 다음과 같이 배열로 받으면 된다. 그러면 배열이 생기는 게 아니라, resultA, resultB 변수가 각각 새로 생긴다.

질문.

a와 b가 함수 showResult() 에 들어가서 변경되었다.

line1, line2 의 a, b는 바뀌었을까?

결과:

```
[LOG]: "
      a: 3
      b: 4
"
-----
[LOG]: "
      a: 1
      b: 2
"
```

첫번째 로그는 resultA, resultB의 결과이며, 두번째 로그는 a, b의 결과이다. 보다시피, 함수 안에서 값을 변경해도 원래 값은 전혀 영향이 없다.

이것을 이해하려면,

함수 바깥에서 선언한 a, b와,

함수 안에서 쓰는 a, b가

서로 다른 a, b라고 생각해야한다.

함수 바깥에서 선언한 a, b : 전역변수(global variable)

함수 showResult() 에서 쓰인 a, b : 지역변수(local variable)

```
1  const a = 1;   전역변수
2  const b = 2;
3
4  function showResult(a: number, b: number) {
5      a = 3;      지역변수
6      b = 4;
7      return [a, b];
8  }
9
10 const [resultA, resultB] = showResult(a, b);
11
12 console.log(`
13     a: ${resultA}
14     b: ${resultB}
15 `);
16
17 console.log(`
18     a: ${a}   a: 1
19     b: ${b}   b: 2
20 `);
```

지역변수부터 알아보자. 함수로 변수가 들어가면, 그대로 들어가는 게 아니라 값이 복사되는 것이며, 함수 안에서 쓰는 ‘지역변수’가 된다. 지역변수를 바꾼다고 해서 원래 전역변수였던 값이 바뀌진 않는다.

전역변수에 대해 알아보자. 전역변수는 코드 어디에서나 다 쓸 수 있다. 함수 안에서 선언한 적이 없어도 쓸 수 있다.



```

1  const a = 1;
2  const b = 2;
3  const c = 3;
4
5  function showResult(a: number, b: number) {
6      a = 3;
7      b = 4;
8      console.log(c);
9      return [a, b];
10 }
11
12 const [resultA, resultB] = showResult(a, b);
13
14 // console.log(`
15 //     a: ${resultA}
16 //     b: ${resultB}
17 // `);
18
19 // console.log(`
20 //     a: ${a}
21 //     b: ${b}
22 // `);

```

line8                    함수 바깥에 있는 c를 찍어내기. 함수가 실행되면 보일 것이다.

결과:

```
[LOG]: 3
```

line8 에서의 console.log(c)의 결과가 찍힌 것으로 보아, 함수 안에서 c를 선언한 적이 없어도, c를 함수 안에서 쓸 수 있다는 것을 알았다. 전역변수이기 때문이다.

그러나 함수 안에서 변경할 수는 없다.

```

1  const a = 1;
2  const b = 2;
3  const c = 3;
4
5  function showResult(a: number, b: number) {
6      a = 3;
7      b = 4;
8      c = 5;
9      return [a, b];
10 }
11
12 const [resultA, resultB] = showResult(a, b);
13
14 // console.log(`
15 //     a: ${resultA}
16 //     b: ${resultB}
17 // `);
18
19 // console.log(`
20 //     a: ${a}
21 //     b: ${b}
22 // `);

```

line8: console.log(c) 문장을 c = 5 로 바꾸었다.



에러 내용:

#### Errors in code

Cannot assign to 'c' because it is a constant.

c는 const 로 선언되었기에 변경할 수 없기 때문이다. 만약 const 대신 let 으로 쓰면 변경할 수 있을 것이지만, let을 꼭 써야 할 상황이 아니기 때문에 변경할 수 없다고만 알아두자.

하지만, 함수 안에서 값을 변경해도 바깥의 값까지 바뀌는 예외가 두가지 있다. 객체와 배열이다.

```
1  const menu = ['짜장면', '짬뽕', '탕수육'];
2
3  function showResult(menu: string[]) {
4      menu[0] = '우동';
5      return menu;
6  }
7
8  const result = showResult(menu);
9  console.log(result);
10 console.log(menu);
```

line3: 배열의 타입은 다음과 같이, 타입[] 으로 지정해준다.

결과:

```
[LOG]: ["우동", "짬뽕", "탕수육"]
[LOG]: ["우동", "짬뽕", "탕수육"]
```

첫번째 로그는 showResult(menu) 의 return 값을 받은 result 의 결과다.

두번째 로그는 원래 전역변수였던 menu의 결과다.

둘이 내용이 일치하는 걸 보니, 함수 안에서 바꿨는데도 값이 바뀌었다는 것을 알 수 있다.

객체도 마찬가지다. 다음 코드를 살펴보자.

```

1  interface Info {
2      name: string,
3      age: number
4  }
5
6  const info = {
7      name: '조교행님',
8      age: 28
9  };
10
11 function showResult(info: Info) {
12     info.age = 18;
13     return info;
14 }
15
16 const result = showResult(info);
17 console.log(result);
18 console.log(info);

```

line1-4 에서 interface라는 새로운 개념이 나왔는데, 이것은 이 수업의 맨 마지막인 interface에서 자세히 다룬다. 지금은 이 부분이 필요하다는 것만 알아두자.

결과:

```

[LOG]: {
  "name": "조교행님",
  "age": 18
}
-----
[LOG]: {
  "name": "조교행님",
  "age": 18
}

```

첫번째 로그는 showResult(info) 의 return 값을 받은 result 의 결과다.

두번째 로그는 원래 전역변수였던 객체 info의 결과다.

이상하다. 분명 지역변수가 되면 함수 바깥에 있는 값이 영향을 안 받는다 하지 않았는가?  
 하지만 **배열과 객체는 예외다. 배열과 객체는 함수 안으로 끌고 와도 전역변수이다.** 값  
 복사가 아니라, 주소가 그대로 들어가기 때문이다. 주소에 대한 설명까지 하면 지나치게  
 어려우므로, 생략하도록 하겠다.

심지어, 함수 안에서 다른 임시 변수를 선언해 복사하더라도 소용없다.

```

1  const menu = ['짜장면', '짬뽕', '탕수육'];
2
3  function showResult(menu: string[]) {
4      const tempMenu = menu;
5      tempMenu[0] = '우동';
6      return tempMenu;
7  }
8
9  const result = showResult(menu);
10 console.log(result);
11 console.log(menu);

```

line4: ‘임시’ 라는 뜻을 가진 temporary 의 줄임말인 temp 를 menu 앞에 붙인 새로운 변수를 만들어 복사했다. temp는 변수 이름 정할 때 자주 쓴다.

결과:

```

[LOG]: ["우동", "짬뽕", "탕수육"]
[LOG]: ["우동", "짬뽕", "탕수육"]

```

우리는 tempMenu 를 새로 만들어 복사했다고 생각했으나, 사실은 복사가 아니고 같은 주소를 가리키는 tempMenu와 menu 두 개가 존재할 뿐이다.

초보자가 이해하긴 굉장히 어려운 내용이다. 나는 여러분의 지식을 늘려주고싶어서 이런 걸 설명한 게 아니다. 단지, **배열과 객체는 함수 안으로 끌고 와도 전역변수이므로, 함수의 인자로 줄 때 매우 조심해서 써야한다는 것만 기억하자.**

\* 지금까지 배운 것에서 알 수 있는 사실은 뭘까? **코드에서 같은 이름으로 쓰였다 하더라도, 어디에 있느냐에 따라서 사실은 완전히 다른 것일 수 있다는 것이다.** Result 와 result 처럼, 대문자와 소문자 하나 차이는 당연히 다른 것을 의미하고, result와 result 처럼 같은 이름을 쓰더라도, 함수 안에 들어가있는지 아닌지에 따라 같은 result 일수도 있고, 다른 result 일수도 있다.

### 3. return 타입 지정 - TS Only

TypeScript에선 선택사항으로 return 타입을 지정할 수 있다.

```
1  function sayHello(name: string, age: number): string {
2      |      return `하이, ${name}. 연세는 ${age}세`;
3      }
4
5  const a = sayHello('조교행님', 28);
6  const b = sayHello('은지', 16);
7  const c = sayHello('현수', 21);
8  const d = sayHello('지원', 14);
9  const e = sayHello('준수', 23);
10
11  console.log(`
12      ${a}
13      ${b}
14      ${c}
15      ${d}
16      ${e}
17  `);
```

함수의 소괄호 뒤에, : 찍고 타입을 써주면 된다.

만약 string이 아닌 다른 타입을 return 하면 무슨 일이 일어날까?

```
1  function sayHello(name: string, age: number): number {
2      |      return `하이, ${name}. 연세는 ${age}세`;
3      }
4
5  const a = sayHello('조교행님', 28);
6  const b = sayHello('은지', 16);
7  const c = sayHello('현수', 21);
8  const d = sayHello('지원', 14);
9  const e = sayHello('준수', 23);
10
11  console.log(`
12      ${a}
13      ${b}
14      ${c}
15      ${d}
16      ${e}
17  `);
```

에러:

## Errors in code

```
Type 'string' is not assignable to type  
'number'.
```

string 타입은 number 타입에 assignable 아니다.

assignable 은 '할당할 수 있는' 이란 뜻이다.

string 타입은 number 타입에 할당할 수 없다.

즉, string 타입만 return 해야한다고 우리가 강제하는 것이다. 안전한 개발을 하고 싶다면, 선택사항이라도 return 타입을 지정해 주는 것이 좋다.

4. parameter와 return 은 필수사항이 아니다.

경우에 따라선, parameter 또는 return 이 필요없는 함수를 만들 수도 있다. 이 경우, parameter와 return 타입은 '빈 공간' 이라는 뜻인 void 가 된다.

다음 코드를 보자.

먼저 parameter가 없는 경우다.

```
1 function sayHello() {  
2   |   return 'Hello';  
3 }  
4  
5 console.log(sayHello());
```

즉, 아무것도 안 받을땐 그냥 소괄호 ( ) 만 써주면 된다.

다음은 return 이 없는 경우다.

```
1 function sayHello(name: string, age: number): void {  
2   |   console.log(`안녕, ${name}. 나이는 ${age}`);  
3 }  
4  
5  
6 sayHello('조교행님', 28);  
7 sayHello('은지', 16);  
8 sayHello('현수', 21);  
9 sayHello('지원', 14);  
10 sayHello('준수', 23);
```

결과:

```
[LOG]: "안녕, 조교행님. 나이는 28"  
[LOG]: "안녕, 은지. 나이는 16"  
[LOG]: "안녕, 현수. 나이는 21"  
[LOG]: "안녕, 지원. 나이는 14"  
[LOG]: "안녕, 준수. 나이는 23"
```

보다시피, `return` 이 없고 `console.log()` 로 결과를 출력시킨다.

또한 아까 설명했듯이 TypeScript 에서 `return` 타입을 지정하는것은 선택사항이므로, `:void` 는 생략 가능하다.

이번 장에서 설명한 내용들은 사실 상당히 어려운 내용이다. 특히 전역변수와 지역변수는 컴퓨터공학과 전공생들도 다른 언어와 비교해가면서 심도 깊게 배우며, 함수 안에 함수가 쓰였을 때는 어떤 식의 결과가 나오는지까지 알아야 한다. 그러나 어렵더라도 함수에 대해 알아야 할 걸 모르고 간다면, 나중에 실전개발에서 문제가 발생했을 때 왜 그렇게 되는지를 찾는 데 한참 걸린다. 이 수업은 노베이스를 위한 수업이 맞지만, React 실전개발을 위해 알아야 할 걸 모르고 넘어가진 않는다.

다음 장에서부터, 우린 더이상 `function` 키워드를 써서 함수를 만들지 않을 것이다. 함수를 만드는 다른 방법이자, 요즘의 방식인 화살표 함수(`arrow function`) 에 대해 배운다.

핵심정리:

지역변수: 함수 안에서 복사해 쓰는 변수. 함수 바깥에 영향을 미치지 않는다.

전역변수: 함수 바깥에서 선언한 변수. 함수 안에서 선언한 적이 없어도 함수 안에서 쓸 수 있다.

그러나, 객체와 배열은 함수 안으로 가져오더라도 전역변수이므로, 함수의 인자로 쓸 땐 매우 조심해야한다.

코드에서 같은 이름으로 쓰였더라도 사실은 완전히 다른 것을 의미할 수 있다.

TypeScript에선 선택사항으로 return 의 타입을 지정할 수 있다. 안전한 개발을 하고 싶다면, 선택사항이라도 return 타입을 지정해 주는 것이 좋다.

경우에 따라선, parameter 또는 return 이 필요없는 함수를 만들 수도 있다. 이 경우, parameter, return 타입은 '빈 공간' 이라는 뜻인 void 가 된다.

## 9. 화살표 함수(arrow function)

지금부터 function 키워드를 쓰지 않고 새로운 방식으로 함수를 만들 것이다. 아니 그럼 대체 function 을 이용한 함수 만드는 법은 왜 배웠단 말인가?

function이 더는 안 쓰이는 옛날 기술이라면 아예 가르치질 않았을 것이다. function으로 함수를 만드는 건 아직도 쓰이는 기술이다. arrow function 은 업그레이드 버전일 뿐이다. 지금까지 함수를 힘들게 배웠으면 arrow function 도 금방 이해할 수 있을 것이다.

코드를 살펴보자. 저번 장에선 이렇게 함수를 만들었다.

```
1 function sayHello(name: string, age: number) {  
2   |   return `안녕, ${name}. 나이는 ${age}세`;   
3   | }  
4  
5 const result = sayHello('조교행님', 28);  
6  
7 console.log(result);
```

화살표 함수(arrow function)를 이용한다면 다음과 같이 바뀐다.

```
1 const sayHello = (name: string, age: number) => {  
2   |   return `안녕, ${name}. 나이는 ${age}세`;   
3   | }  
4  
5 const result = sayHello('조교행님', 28);  
6  
7 console.log(result);
```

결과:

```
[LOG]: "안녕, 조교행님. 나이는 28세"
```

차이점을 비교해보자.

1. function 키워드가 사라졌다.
2. 함수를 변수 선언하듯이 const 로 선언했고, = 를 통해 값을 집어넣는다.
3. => 이라는 화살표가 생겼다.

핵심은, 함수가 변수가 되었다는 것이다.

JavaScript에서는 함수는 변수이며, 하나의 자료형이다.

함수가 변수이기 때문에, 다른 함수의 parameter 로도 쓰이고, return 으로도 쓰인다.

화살표 함수는 JavaScript에서 함수의 정체를 확실히 보여준다. sayHello라는 변수에



집어넣는다는 뜻은, 변수에 들어가기전에 함수 역시 의미있는 존재, 람다식(lambda function)임을 말해준다. 람다식은 쉽게 말해서, 이름 없는 함수다.

\* lambda 는 그리스어 알파벳 람다(  $\lambda$  ) 를 의미한다. 프로그래밍에서 lambda는 미적분학에서의 lambda에서 유래되었다. 그렇다고 lambda function을 이해하는 데 미적분을 꼭 알아야하는것은 아니다.

arrow function 을 사용하면 대표적으로 좋은 점은 다음과 같다.

1. function 키워드로 만드는 함수에 비해 문법이 간단하다.
2. 간단하게 쓰고 버릴 함수를 function 키워드로 선언할 필요 없이 금방 만들 수 있다.
3. 함수형 프로그래밍을 좀 더 쉽게 만든다.

지금부터 말하는 내용은 어려운 개념이고, 당장 필요하지도 않기에 일부러 주황색으로 썼다. 읽고 싶지 않으면 읽지마라.

솔직히 2번과 3번은 무슨 말인지 와닿지 않을 것이다. 여러분은 지금까지 함수형 프로그래밍을 배운 적이 없기 때문이다. 다음 강의인 React에선 함수형 프로그래밍인 Hook 을 일상적으로 쓸 것이기 때문에, 연습하다가 익숙해질 수 있다. 그래서, 2번과 3번을 먼저 체험하게 해주겠다.

함수형 프로그래밍은 다음과 같은 게 가능하다.

```
1  const adder = (x: number) => (y: number) => x + y;
2  const add5 = adder(5);
3  console.log(add5(1));
```

결과:

```
[LOG]: 6
```

line1 에서 잘 보면, 우리가 이제까지 배운 함수는 (y: number) 에서 끝났어야했다. 그러나 adder의 return 값은 (y: number) 가 아니라, (y: number) => x + y; 라는 람다식, 이름 없는 함수이다. 보다시피, 함수의 return 이 함수다.

line2와 line3에서 좀 더 확실히 알수 있다. adder(5) 의 결과는 add5() 라는 새로운 함수가 되어, 5를 x에 집어넣는다. 즉, 다음과 같은 식이 된다.

```
5  const add5 = (y: number) => 5 + y;
6  console.log(add5(1));
```

\* adder() 함수의 지역변수인 x를 사용하는 함수 안의 함수 (y: number) => x + y; 를 클로저(closure)라고 한다. 꽤 어려운 개념이니 클로저를 경험했다고만 생각하고 넘어가자.

세 줄로 끝날 이 코드를 쓰기 위해, 화살표 함수가 없던 시절에는 함수를 function 키워드로 하나하나 선언했어야했다. 화살표 함수를 통해 간단하게 쓰고 버릴 함수를 쉽게 만들 수 있다.

엄청 어려운 부분이 끝났다. 아마 이 강의 중 가장 어려운 부분이었을 것이다. 직접 이런 문제에 부딪혀본적이 없기 때문이다. 그렇다면 문법이 간단해진다는 1번부터 경험해보자. 덧셈하는 코드이다.

```
1  const add = (a: number, b: number) => {
2    |    return a + b;
3  };
4
5  console.log(add(5, 2));
```

결과:

```
[LOG]: 7
```

이 코드를 훨씬 간단히 줄일 수 있다. 중괄호 { } 안에 들어가는 코드는 오직 return 을 나타낸 한 문장이다. 그래서 **중괄호를 생략하고 return 도 생략**해서 다음과 같이 쓸 수 있다.

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

연습으로, 계산기를 만들어보자. function 키워드를 써서 먼저 만들어보겠다.

```
1  const myCal = {
2    |    add: function(a: number, b: number) {
3    |      |    return a + b;
4    |    },
5    |    subtract: function(a: number, b: number) {
6    |      |    return a - b;
7    |    },
8    |    multiply: function(a: number, b: number) {
9    |      |    return a * b;
10   |    },
11   |    devide: function(a: number, b: number) {
12   |      |    return a / b;
13   |    }
14 };
15
16 console.log(myCal.add(5, 2));
17 console.log(myCal.subtract(10, 3));
18 console.log(myCal.multiply(10, 2));
19 console.log(myCal.devide(30, 2));
```

\* 함수 역시 자료형이기때문에, 당연히 객체의 프로퍼티로 쓸 수 있다. 이렇게 객체 안에서 정의된 함수를 메소드(method) 라고 한다.

결과:

```
[LOG]: 7
```

```
[LOG]: 7
```

```
[LOG]: 20
```

```
[LOG]: 15
```

화살표함수를 쓰면 다음과 같이 줄일 수 있다.

```
1  const myCal = {
2    add: (a: number, b: number) => a + b,
3    subtract: (a: number, b: number) => a - b,
4    multiply: (a: number, b: number) => a * b,
5    devide: (a: number, b: number) => a / b
6  };
7
8  console.log(myCal.add(5, 2));
9  console.log(myCal.subtract(10, 3));
10 console.log(myCal.multiply(10, 2));
11 console.log(myCal.devide(30, 2));
```

이처럼 문법이 간단해졌다.

주의: 중괄호 { } 와 return을 생략할 경우, 객체를 return 할 때는 꼭 소괄호 ( ) 로 감싸줘야 한다. 함수에서 중괄호 { } 는 함수 내용을 의미하기 때문이다.

```
1  const test = () => ({ job: '개발자' });
```

화살표 함수를 이 정도로 하고 끝내겠다. 다음에 배울 것은 반복적인 작업을 할 때 사용하는 map() 과 filter() 이다.

핵심정리:

JavaScript에서는 함수는 변수이며, 하나의 자료형이다.

함수가 변수이기 때문에, 다른 함수의 parameter 로도 쓰이고, return 으로도 쓰인다.

return 이 한 줄인 함수의 경우, 중괄호를 생략하고 return 도 생략 가능하다.

중괄호 { } 와 return을 생략할 경우, 객체를 return 할 때는 꼭 소괄호 ( ) 로 감싸줘야 한다. 함수에서 중괄호 { } 는 함수 내용을 의미하기 때문이다.

## 10. 반복 - map() 과 filter()

사람이 컴퓨터를 쓰는 가장 큰 이유 중 하나는, 귀찮은 반복작업을 빠르게 처리한다는 것이다. 그래서 다른 강의나 책에선 컴퓨터에게 반복을 시키는 두가지 방법, while 과 for 에 대해 꼭 다루고 넘어간다. 그러나 이 강의에선 while 과 for를 다루지 않는다. while과 for는 어떤 언어에서든 비슷한 문법을 가지고 있으므로(파이썬 제외), 궁금하면 다른 사람의 자료를 참고해보면 될 것이다. JavaScript에서도 당연히 while 과 for 를 쓸 수 있지만, 우리 map() 과 filter() 를 사용해 반복을 해볼 것이다.

보통 반복문 강의는 if else 다음에 나오는데 이 책에는 꽤 후반부에 나왔다. map() 과 filter() 는 콜백함수라서 반드시 함수 다음에 설명해야되기 때문이다.

콜백함수(callback function)는 함수 안에 ‘들어와서’ call 되는 함수이며, 함수 안에 있는 함수이다. 안에 있는 함수의 작동이 끝나야만 바깥에 있는 함수가 처리된다.

map() 을 바로 살펴보자.

```
1  const students = ['조교행님', '지연', '혁주', '민지', '은혁'];
2
3  const newStudents = students.map(student => `${student} 최고`);
4
5  console.log(newStudents);
```

결과:

```
[LOG]: ["조교행님 최고", "지연 최고", "혁주 최고", "민지 최고", "은혁 최고"]
```

이론적인 걸 어렵게 설명하기전에, 간단히 생각해보자. 배열 students(복수. s 사용) 에 map() 을 사용했더니, 새로운 배열 newStudents 가 나왔다.

map() 은 JavaScript 내장 객체인 Array의 함수다. 내가 JavaScript에서 선언하는 모든 배열은 Array 객체다. 그래서 배열 students 에서 우리가 만든 적도 없는 map() 함수를 사용할 수 있는 것이다.

line3 에서 자세히 보면, map 안에 인자로 뭐가 들어가는가? 함수가 들어간다. 지난 장에서 설명했듯이, JavaScript에선 함수가 변수이기 때문에, 다른 함수의 parameter 로도 쓰이고, return 으로도 쓰인다. 즉, map() 의 소괄호 안에는 함수만 넣어야 한다.

그리고 student(단수. s안쓰임) 도 사실 student() 라는 함수이며, 콜백함수다. student()는 매개변수로 배열의 각각의 요소를 받으며, map() 은 student() 라는 콜백함수를 배열의 각각의 요소마다 실행시킨다.

student('조교행님')		'조교행님 최고'
student('지연')		'지연 최고'
student('혁주')	return	'혁주 최고'
student('민지')		'민지 최고'
student('은혁')		'은혁 최고'

그리고 이 결과들로 새 배열을 만들어서 return 한다. 새 배열의 내용은 다음과 같다.

```
[LOG]: ["조교행님 최고", "지연 최고", "혁주 최고", "민지 최고", "은혁 최고"]
```

우리는 그 결과를 newStudents 로 받았다.

map() 함수가 어떻게 생겼는지, 정확히 어떤 식으로 작동하는지를 변수 하나하나 따라가면서 설명하지 않는다. 중요한 건, **map() 은 배열을 반복해 새 배열로 바꿔서 return 한다**는 것이다.

다음은 filter() 다. 우리가 흔히 쓰는 단어인 필터는 뭔가를 걸러낼 때 쓰는 것이다. 학생 이름 배열에서 이름이 긴 사람인 '조교행님' 만 걸러내보겠다.

```
1 const students = ['조교행님', '지연', '혁주', '민지', '은혁'];
2
3 const newStudents = students.filter(student => student.length > 2);
4
5 console.log(newStudents);
```

결과:

```
[LOG]: ["조교행님"]
```

이번엔 다섯개가 아니라 하나만 만들어졌다. 아까랑 비슷한데, 함수의 return 값에 '조건'이 들어갔다. 길이가 2 초과여야 한다는 것이다! 반복을 하면서 이 조건을 일일이 검사한 후, 조건에 맞는 것으로 새 배열을 만든다.

student('조교행님')	'조교행님'.length > 2	true
student('지연')	'지연'.length > 2	false
student('혁주')	'혁주'.length > 2	false
student('민지')	'민지'.length > 2	false
student('은혁')	'은혁'.length > 2	false

return ['조교행님']

컴퓨터의 목적 중 하나인 반복까지 배웠으면, 여러분은 진정한 프로그래머다. 다음 장이 드디어 마지막이다. TypeScript에서 객체를 parameter로 받을 땐 꼭 해줘야 할 작업이 있다. interface 다.

핵심정리:

콜백함수(callback function)는 함수 안에 ‘들어와서’ call 되는 함수이며, 함수 안에 있는 함수이다.

콜백함수는 안에 있는 함수의 작동이 끝나야만 바깥에 있는 함수가 처리된다.

map() 과 filter() 는 parameter로 함수를 받는다.

map() 은 배열을 반복해 새 배열로 바꿔서 return 한다.

filter() 는 배열을 반복해 조건에 맞는것만 걸러낸 후, 새 배열을 만들어 return 한다.

## 11. interface - TS Only

드디어 JavaScript, TypeScript 강의의 마지막 장인 interface 다. 이 문법은 JavaScript엔 없고, TypeScript 에만 존재한다. 그래서 제목부터가 아예 파란색이다. 서론에서 말했듯, TypeScript 가 나오는 부분은 파란색으로 설명한다고 했는데, 장 전체를 파란색으로 칠할 순 없지 않나?

\* 이 강의는 react로 점프하기위한 강의이기 때문에, interface중에서 객체 interface만 다룬다. 변수 interface, 함수 interface, 클래스 interface는 이 강의에서 다루지 않는다.

다음 코드를 일단 살펴보자

```
1  const student = {  
2    name: '조교행님',  
3    age: 28,  
4    gender: '남성'  
5  };  
6  
7  const sayHello = (student: object): string => {  
8    return `안녕, ${student.name}. 나이는 ${student.age}세. 성별은 ${student.gender}`;  
9  };  
10  
11 console.log(sayHello(student));
```

에러 내용:

### Errors in code

- Property 'name' does not exist on type 'object'.
- Property 'age' does not exist on type 'object'.
- Property 'gender' does not exist on type 'object'.

대체 뭐가 잘못된걸까? student의 타입은 확실히 object 가 맞다. 그러나, TypeScript는 이를 허용하지 않는다. TypeScript에선, object를 받을 땐 정확하게 어떻게 생겼는지 써줘야한다. 즉, 반드시 interface를 정의해줘야 한다.

interface는 쉽게 말하면 내가 만든 타입이다. 우리가 지금까지 배웠던 타입은 뭐가 있었나? number, string, boolean, array, object, function 이었다. 우린 우리만의 타입을 만들어 볼 것이다. 예시를 바로 보자!



```

1 interface Student {
2   name: string,
3   age: number,
4   gender: string
5 }
6
7 const student = {
8   name: '조교행님',
9   age: 28,
10  gender: '남성'
11 };
12
13 const sayHello = (student: Student): string => {
14   return `안녕, ${student.name}. 나이는 ${student.age}세. 성별은 ${student.gender}`;
15 };
16
17 console.log(sayHello(student));

```

여기서, 원래 저 부분은 object였는데 이젠 Student로 바뀌었다. 즉, Student 타입이다. number, string, boolean, array, object, function 이 아니라, 우리가 line1-5 에서 만들어준 Student 타입이다.

JavaScript에선 object가 맘대로 들어갈 수 있도록 했다. 하지만 TypeScript에서는 허락하지 않는다. 이렇게 하면 뭐가 좋아지는지 알아보자. 일부러 에러를 일으켜보는게 최고다.

상황1.

```

1 interface Student {
2   name: string,
3   age: number,
4   gender: string
5 }
6
7 const student = {
8   named: '조교행님',
9   age: 28,
10  gender: '남성'
11 };
12
13 const sayHello = (student: Student): string => {
14   return `안녕, ${student.name}. 나이는 ${student.age}세. 성별은 ${student.gender}`;
15 };
16
17 console.log(sayHello(student));

```

프로퍼티 키 이름인 name을 named 로 잘못 썼을 경우다. 상식적으로 당연히 에러가 날거라고 생각하지 않는가? line14에선 name으로 쓰고 있기 때문에, 이대로면 없는 걸 가져오는게 되므로 에러다. 그러나 우습게도, JavaScript는 이 코드를 정상으로 받아들여 다음을 출력한다.

```
[LOG]: "안녕, undefined. 나이는 28세. 성별은 남성"
```

이 문장은 JavaScript 입장에선 정상적인 결과다. 이런 종류의 ‘문제’는 웹개발하는 사람들에게 가장 많이 발생하고, 원인을 찾기도 힘들다. 에러메세지를 안 띄워주기 때문이다. 자고로 개발자한테 가장 무서운 상황은, 분명히 우리한테 에러인데 에러 문구도 없이 멀쩡하게 돌아가는 경우다. 개발자는 눈에 불을 켜고 코드를 뒤져가면서, 어디까지 데이터가 들어왔나 `console.log()` 입력해가면서 찾아야한다. 그리고 결국 찾은 이유가 이런 사소한 철자 오류 때문이었다고 생각해보자. 얼마나 힘이 빠지는가?

TypeScript는 이 경우엔 에러메세지를 띄워서 우리의 실수를 재빨리 찾도록 도와준다.

에러 내용:

```
Errors in code
Argument of type '{ named: string; age: number; gender: string; }' is not assignable to parameter of type 'Student'.
Property 'name' is missing in type '{ named: string; age: number; gender: string; }' but required in type 'Student'.
```

문장1. 타입 { ~~~ } 는 할당할 수 없다 / Student 타입의 parameter 에.  
= Student 타입의 parameter에 타입 { ~~~ }는 할당할 수 없다.

문장2는 문장 1을 구체적으로 설명한 것이다.

‘name’ 프로퍼티가 타입 { ~~~ }에서 빠졌다 / 그러나 Student 타입엔 필요하다.

즉, 함수에 집어넣으려면 student에 name이 꼭 필요한데, 못찾겠다는 것이다. 우리가 잘못 썼기 때문이다. TypeScript 덕분에 훨씬 빨리 문제를 찾을 수 있지 않았나?

상황2.

```
1 interface Student {
2   name: string,
3   age: number,
4   gender: string
5 }
6
7 const student = {
8   name: '조교행님',
9   age: 28,
10  gender: '남성'
11 };
12
13 const sayHello = (student: Student): string => {
14   return `안녕, ${student.name}. 나이는 ${student.age}세. 성별은 ${student.sex}`;
15 };
16
17 console.log(sayHello(student));
```

이번엔 gender를 실수로 sex로 적은 경우다. 둘 다 성별을 의미하는 단어이니 이처럼 실수할 경우도 있다. 이 경우에도 당연히 에러가 날 것이라고 기대하지만, JavaScript에선 이게 에러가 아니며, 정상작동해 다음과 같은 결과를 출력한다.

```
[LOG]: "안녕, 조교행님. 나이는 28세. 성별은 undefined"
```

그러나 TypeScript는 에러메세지를 재빨리 띄워주어 우리의 실수를 막는다.

에러 내용:

#### Errors in code

```
Property 'sex' does not exist on type 'Student'.
```

프로퍼티 sex는 타입 Student에 존재하지 않습니다.

이게 상식적이다. 꼭 우리가 만든 Student 타입을 생각하지 않더라도, 변수 student 객체엔 sex가 없지 않는가?

상황3.

```
1 interface Student {  
2   name: string,  
3   age: number,  
4   gender: string  
5 }  
6  
7 const student = {  
8   name: '조교행님',  
9   age: 28,  
10  gender: '남성',  
11  phone: "010-5806-8112"  
12 };  
13  
14 const sayHello = (student: Student): string => {  
15   return `안녕, ${student.name}. 나이는 ${student.age}세. 성별은 ${student.gender}`;  
16 };  
17  
18 console.log(sayHello(student));
```

객체 student에 전화번호를 추가한 경우다. 이 경우, 에러는 TypeScript일지라도 발생하진 않는다.

하지만, 이 상태에서 실제로 전화번호를 쓰려고 하면 문제가 된다.

```

1 interface Student {
2   name: string,
3   age: number,
4   gender: string
5 }
6
7 const student = {
8   name: '조교행님',
9   age: 28,
10  gender: '남성',
11  phone: "010-5806-8112"
12 };
13
14 const sayHello = (student: Student): string => {
15   return `안녕, ${student.name}. 나이는 ${student.age}세. 성별은 ${student.gender}, 전화번호는 ${student.phone}`;
16 };
17
18 console.log(sayHello(student));

```

에러 내용:

### Errors in code

Property 'phone' does not exist on type 'Student'.

'phone' 프로퍼티는 존재하지 않습니다 / Student 타입에  
= Student 타입에 phone 프로퍼티는 존재하지 않습니다.

우리가 Student 타입을 만들 때 phone 프로퍼티를 만들어놓지 않았기 때문에 에러가 발생했다. 이 경우엔 이런 불만이 생길 수 있다.

우리가 phone을 좀 넣어서 쓸수도 있지, TypeScript가 너무 까다로운 것 아닌가?

그러나, student 객체 변수가 하나가 아니라 student1, student2, student3처럼 여러개를 사용한다면, 어떤 건 phone이 있고, 어떤건 phone이 없는채로 써버리는 제멋대로인 상황이 될 수도 있다. 상식적으로, student1, 2, 3은 모두 똑같은 형태를 가지고 있어야 하지 않을까? 우리가 아예 interface를 써서 Student 타입을 만들어줬다면 나중에 생길수도 있는, 혹시 모를 혼란을 방지해준다.

상황4.

```
1 interface Student {
2     name: string,
3     age: number,
4     gender: string
5 }
6
7 const student = {
8     name: '조교쌤',
9     age: '28',
10    gender: '남성'
11 };
12
13 const sayHello = (student: Student): string => {
14     return `안녕, ${student.name}. 나이는 ${student.age}세. 성별은 ${student.gender}`;
15 };
16
17 console.log(sayHello(student));
```

Student 타입의 age는 number인데, 변수 student 의 age는 string이다. TypeScript에선 다음과 같은 에러가 발생한다.

에러 내용:

```
Errors in code

Argument of type '{ name: string; age: string; gender: string; }' is not assignable to parameter of type 'Student'.
Types of property 'age' are incompatible.
Type 'string' is not assignable to type 'number'.
```

Student 타입의 parameter에 타입 { ~~~ }는 할당할 수 없다.

프로퍼티의 타입 '들' age 프로퍼티는 호환성이 없다. (incompatible)

= age 프로퍼티끼리 안 맞다.

string 타입은 number 타입에 할당할 수 없다.

이 상황 역시 같은 용도로 쓸 객체는 같은 형태를 가져야한다고 TypeScript가 잔소리하는 것이다. 상황3과 같이, 미래에 생길수도 있는 문제를 방지해서 결과가 깔끔하게 나오도록 돕는다.

정리해서, 함수의 인자로 객체를 줄 때 interface를 강제하면 뭐가 좋은걸까? 수많은 문제를 미리 방지할 수 있을 것이다. 강의의 OT에서 말했듯이, 중국집의 메뉴판대로만 주문받는 카운터 직원을 고용한 것이나 다름없다. 메뉴판이 무엇인가? 장사 시작할 때부터 우리가 정해놔서, 당연히 그것만 주문 올거라고 믿어 의심치 않는 것이다. TypeScript는 누군가 중국집에 메뉴판에도 없는 스파게티를 시켰을 때, 주방에 그대로 주문해서 주방에 혼란을 주지 않는다. 심지어, 아무 내용도 없는 주문(undefined)을 하지도 않는다. 주문이 들어오면 진짜 메뉴판에 있는 것과 일치하는지 반드시 확인한다.

TypeScript는 이처럼 에러메세지도 없이 문제를 찾는 불쌍한 웹개발자들의 수명을 늘려주는게 확실하다.

핵심정리:

interface는 JavaScript엔 없고, TypeScript 에만 있는 개념이다

interface는 직접 만든 타입이다

함수의 인자로 객체를 쓸때는, 반드시 객체를 interface로 정의해야한다

이것으로, React를 배우기 위한 기초 JavaScript, TypeScript 를 모두 다루었다. 사실 지금까지 한 내용이 쉽진 않았다. 나는 분명 노베이스도 들어도 된다 그랬지 강의 자체가 쉽다고는 안했다. 깊게 다뤄야 될 부분은 꽤 깊게 다루어서 고생 좀 했을 것이다. 그렇다고 너무 어렵게 다루지도 않았다. 지금까지 다루었던 개념들이 JavaScript, TypeScript의 전부는 아니다. 다른 JavaScript 기본서에서 전부 다루는 prototype (신문법 class)은 아예 포함도 안 시켰다. 즉, 객체지향 부분을 아예 제외했다. React가 Hooks 를 지원하면서 진입장벽이 많이 낮아졌기 때문에 class를 몰라도 개발하는데 지장이 없기 때문이다. 만약 class 가 필요하다면, prototype을 마주한다면, 그때 공부하면 된다. 모든 걸 다 하고 다음으로 가려고 하면 평생 개발 못한다.

이제 우린 console 창에서 벗어날 것이다. 진짜 웹개발의 세계로, 진짜 화면을 만들러 가보자. 다음 강의는 React Hook 이다.