ES6 고급 & typescript

학습목표

javascript ES6 고급 문법

typescript 문법 기초

목차

[1. let, var, const 2](#_Toc130040337)

[1) let 키워드로 변수 선언 2](#_Toc130040338)

[2) var 키워드로 변수 선언 3](#_Toc130040339)

[2. 함수 7](#_Toc130040340)

[1) default parameter 7](#_Toc130040341)

[2) 가변 파라미터 함수 9](#_Toc130040342)

[3. 구조 분해 할당 (destructuring assignment) 11](#_Toc130040343)

[1) 배열 구조 분해 할당 11](#_Toc130040344)

[2) 객체 구조 분해 할당 13](#_Toc130040345)

[4. 전개 연산자 (spread operator) 15](#_Toc130040346)

[1) 배열 분해 15](#_Toc130040347)

[2) 객체 분해 17](#_Toc130040348)

[5. module 18](#_Toc130040349)

[1) 모듈(module)이란? 18](#_Toc130040350)

[2) package.json 파일 수정 18](#_Toc130040351)

[3) export & import 19](#_Toc130040352)

[4) export default 20](#_Toc130040353)

[5) 하나씩 export 22](#_Toc130040354)

[6. 함수와 화살표 함수 23](#_Toc130040355)

[1) 메소드와 this 23](#_Toc130040356)

[2) inner 함수 25](#_Toc130040357)

[3) 화살표 함수 26](#_Toc130040358)

[4) 요약 28](#_Toc130040359)

[7. 배열의 메소드 29](#_Toc130040360)

[8. typescript 기초 30](#_Toc130040361)

[1) typescript 설치 30](#_Toc130040362)

[2) 프로젝트 생성 30](#_Toc130040363)

[3) 변수 선언 32](#_Toc130040364)

[4) 타입의 종류 33](#_Toc130040365)

[5) 배열 34](#_Toc130040366)

[6) 함수 선언 35](#_Toc130040367)

[7) 타입 별칭 (type alias) 36](#_Toc130040368)

[8) 열거형 38](#_Toc130040369)

# let, var, const

## let 키워드로 변수 선언

### 파일 생성

Graphical user interface, text, application

Description automatically generated

chap01 폴더를 선택하고, 파일 생성 버튼을 클릭하여, chap01 폴더 아래에 hello2.js 파일을 만들자.

### hello2.js

|  |
| --- |
| let message = "hello world";  console.log(message); |

let 키워드를 사용하여, 지역 변수를 선언한다.

javascript 언어에서는 변수를 선언할 때 변수의 타입을 지정하지 않는다.

let message = "hello world";

messsage 지역 변수에 "hello world" 문자열이 대입된다.

### hello2.js 실행

|  |
| --- |
| A picture containing logo  Description automatically generated |

## var 키워드로 변수 선언

### hello3.js

|  |
| --- |
| var message = "hello world";  console.log(message); |

출력

|  |
| --- |
| hello world |

var 키워드를 사용하여 변수를 선언할 수도 있다.

var 키워드로 선언한 변수는, let 키워드로 선언한 변수와 미묘한 차이점이 있다.

let 키워드로 선언한 지역변수는 Java나 C의 지역 변수 문법과 같다.

var 키워드 문법이 특별하다.

### 호이스팅(hoisting)

함수가 호출되면 함수 본문의 실행을 시작하기 전에,

먼저 함수에 var 키워드로 선언된 지역 변수들을 전부 미리 생성한 후 함수 본문을 실행한다.

### var1.js

|  |
| --- |
| function test1() {      var a = 3;      var a = 4;      console.log(a);  }  test1(); |

같은 이름의 변수가 중복 선언되어 있어도 에러가 발생하지 않는다.

출력

|  |
| --- |
| 4 |

### var2.js

|  |
| --- |
| function test2() {      a = 3;      var a = 4;      console.log(a);  }  test2(); |

var 키워드로 선언된 지역 변수들을 전부 미리 생성하기 때문에,

변수를 선언하는 문장 보다 사용하는 코드가 먼저 나와도 괜찮다.

출력

|  |
| --- |
| 4 |

### var3.js

|  |
| --- |
| function test3() {      console.log("A)", sum);      for (let i = 1; i <= 10; ++i) {          var sum = 0;          sum = sum + i;          console.log("B)", sum);      }      console.log("C)", sum);  }  test3(); |

출력

|  |
| --- |
| A) undefined  B) 1  B) 2  B) 3  B) 4  B) 5  B) 6  B) 7  B) 8  B) 9  B) 10  C) 10 |

test3 함수가 호출되자 마자 sum 변수가 미리 만들어지지만, 아직 값이 대입되지 않았다.

따라서 A) 부분에서 sum 변수 값은 undefined 이다.

for 본문에서 sum 변수에 0이 대입된 후, i 변수가 대입된다.

따라서 B) 부분에서 출력되는 값은 0 + i 값이다.

C) 부분에서 sum 변수를 사용할 수 있음에 주목하자.

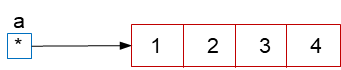
var 키워드로 선언된 모든 변수는 test3 함수 전체에서 접근할 수 있기 때문이다.

### const1.js

|  |
| --- |
| const a = [1, 2, 3, 4];  console.log(a);  a[0] = 10;       // A)  console.log(a);  a = [5, 6];   // B) |

출력

|  |
| --- |
| [ 1, 2, 3 ]  [ 10, 2, 3 ]  C:\PJ\frontend\node1\const1.js:7  a = [2, 3, 4];  ^  TypeError: Assignment to constant variable.  at Object.<anonymous> (C:\PJ\frontend\node1\const1.js:7:3)  at Module.\_compile (node:internal/modules/cjs/loader:1254:14)  at Module.\_extensions..js (node:internal/modules/cjs/loader:1308:10)  at Module.load (node:internal/modules/cjs/loader:1117:32)  at Module.\_load (node:internal/modules/cjs/loader:958:12)  at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run\_main:81:12)  at node:internal/main/run\_main\_module:23:47  Node.js v18.14.2 |



a 변수는 const 이다.

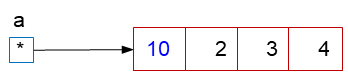
따라서 a 변수에 다른 값을 대입할 수 없다.

그래서 B) 부분에서 에러가 발생한다.



A) 부분에서 값이 대입되는 곳은 배열 내부이지 a 변수가 아니다.

따라서 A) 부분은 에러가 아니다.



**OK**

### const2.js

|  |
| --- |
| const p = { name: "홍길동", age: 16 };  console.log(p);  p.age = 17;   // A)  console.log(p);  p = { name: "임꺽정", age: 19 };   // B) |

출력

|  |
| --- |
| { name: '홍길동', age: 16 }  { name: '홍길동', age: 17 }  C:\PJ\frontend\node1\const2.js:7  p = { name: "임꺽정", age: 19 }; // B)  ^  TypeError: Assignment to constant variable.  at Object.<anonymous> (C:\PJ\frontend\node1\const2.js:7:3)  at Module.\_compile (node:internal/modules/cjs/loader:1254:14)  at Module.\_extensions..js (node:internal/modules/cjs/loader:1308:10)  at Module.load (node:internal/modules/cjs/loader:1117:32)  at Module.\_load (node:internal/modules/cjs/loader:958:12)  at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run\_main:81:12)  at node:internal/main/run\_main\_module:23:47 |

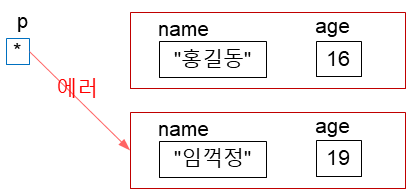
Shape, rectangle

Description automatically generated

p 변수는 const 이다.

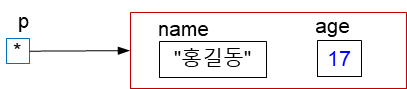
따라서 p 변수에 다른 값을 대입할 수 없다.

그래서 B) 부분에서 에러가 발생한다.



A) 부분에서 값이 대입되는 곳은 객체 내부의 age 속성이지 p 변수가 아니다.

따라서 A) 부분은 에러가 아니다.



**OK**

# 함수

## default parameter

### default1.js

|  |
| --- |
| function printPerson(name, age, email) {      console.log("name=", name, "  age=", age, "  email=", email);  }  printPerson("홍길동", 16, "hong@skhu.ac.kr");  printPerson("홍길동", 16);  printPerson("홍길동");  printPerson(); |

출력

|  |
| --- |
| name= 홍길동 age= 16 email= hong@skhu.ac.kr  name= 홍길동 age= 16 email= undefined  name= 홍길동 age= undefined email= undefined  name= undefined age= undefined email= undefined |

printPerson 함수의 파라미터 변수는 3 개이다.

그런데 printPerson 함수를 호출할 때 파라미터 값을 3개 다 전달하지 않아도 된다.

값이 전달되지 않은 파라미터 변수의 값은 undefined 이다.

### default2.js

|  |
| --- |
| function printPerson(name = "없음", age = 0, email = "없음") {      console.log("name=", name, "  age=", age, "  email=", email);  }  printPerson("홍길동", 16, "hong@skhu.ac.kr");  printPerson("홍길동", 16);  printPerson("홍길동");  printPerson(); |

출력

|  |
| --- |
| name= 홍길동 age= 16 email= hong@skhu.ac.kr  name= 홍길동 age= 16 email= 없음  name= 홍길동 age= 0 email= 없음  name= 없음 age= 0 email= 없음 |

값이 전달되지 않은 파라미터 변수의 디폴트 값 지정하기

### default3.js

|  |
| --- |
| function sum(a = 0, b = 0, c = 0) {      return a + b + c;  }  console.log(sum(1, 2, 3));  console.log(sum(1, 2));  console.log(sum(1));  console.log(sum()); |

출력

|  |
| --- |
| 6  3  1  0 |

## 가변 파라미터 함수

### varargs1.js

|  |
| --- |
| function sum(...numbers) {      let result = 0;      for (let i of numbers)          result += i;      return result;  }  console.log(sum(4, 3, 2, 1));  console.log(sum(3, 2, 1));  console.log(sum(2, 1));  console.log(sum(1));  console.log(sum()); |

출력

|  |
| --- |
| 10  6  3  1  0 |

function sum(...numbers) {

sum 함수에 전달된 파라미터 값들은 전부 배열 한 개로 묶여서,

그 배열이 numbers 파라미터 변수에 전달된다.

즉 numbers 파라미터 변수의 타입은 배열이다.

for (let i of numbers)

numbers 배열의 값들을 하나씩 i 변수에 대입하면서

for 문의 본문을 반복 실행한다.

### varargs2.js

|  |
| --- |
| function sum(numbers) {      let result = 0;      for (let i of numbers)          result += i;      return result;  }  console.log(sum([4, 3, 2, 1])); // OK  console.log(sum(4, 3, 2, 1));   // 에러 |

출력

|  |
| --- |
| 10  error |

sum([4, 3, 2, 1])

sum 함수를 호출하면서, 파라미터 값으로 전달되는 것은 배열이다.

[4, 3, 2, 1] 이 배열이 numbers 파라미터 변수에 전달되어 함수 본문이 실행된다.

OK

sum(4, 3, 2, 1)

sum 함수를 호출하면서, 파라미터 값으로 전달되는 것은 숫자 4 개이다.

이 파라미터 값들 중 첫 번째 값은 4가 numbers 파라미터 변수에 전달되어 함수 본문이 실행된다.

numbers 변수의 값이 배열이 아니고 숫자이기 때문에

for (let i of numbers) 부분에서 에러가 발생한다.

### varargs3.js

|  |
| --- |
| function sum(...numbers) {      let result = 0;      for (let i of numbers)          result += i;      return result;  }  console.log(sum(4, 3, 2, 1));   // OK  console.log(sum([4, 3, 2, 1])); // 에러 |

sum(4, 3, 2, 1)

sum 함수의 파라미터로 전달되는 것은 숫자 4 개이다.

이 4 개의 숫자가 배열로 묶이고, 그 배열이 numbers 파라미터 변수.

예를 들어 numbers[0] 값은 숫자 4 이다.

OK

sum([4, 3, 2, 1])

sum 함수의 파라미터로 전달되는 것은 배열 한 개이다.

이 1 개의 배열이 배열로 묶이고, 그 배열이 numbers 파라미터 변수에 전달된다.

예를 들어 numbers[0] 값은 배열이다. [4, 3, 2, 1]

오류

# 구조 분해 할당 (destructuring assignment)

## 배열 구조 분해 할당

### dassign1.js

|  |
| --- |
| let a = [1, 2, 3];  let x = a[0];  let y = a[1];  let z = a[2];  console.log(x, y, z); |

출력

|  |
| --- |
| 1 2 3 |

### dassign2.js

|  |
| --- |
| let a = [1, 2, 3];  let [x, y, z] = a;  console.log(x, y, z); |

출력

|  |
| --- |
| 1 2 3 |

let x = a[0];

let y = a[1];

let z = a[2];

위 코드와 아래 코드는 완전히 같다.

즉 x, y, z 변수가 생성되고, 그 변수들에a[0], a[1], a[2] 값이 순서대로 대입된다.

let [x, y, z] = a;

let [x, y, z] = a;

x, y, z 변수가 생성되고,

a 배열이 쪼개져서(분해되서), a 변수 내부 값들 각각이 순서대로 x, y, z 변수에 대입된다.

### dassign3.js

|  |
| --- |
| let a = [1, 2, 3];  let x, y, z;  [x, y, z] = a;  console.log(x, y, z); |

출력

|  |
| --- |
| 1 2 3 |

let x, y, z;

x, y, z 변수를 생성한다.

[x, y, z] = a;

x, y, z 변수들에a[0], a[1], a[2] 값이 순서대로 대입된다.

## 객체 구조 분해 할당

### dassign4.js

|  |
| --- |
| let person = { name: "홍길동", age: 16, email: "hong@skhu.net" };  let name = person.name;  let age = person.age;  let email = person.email;  console.log(name, age, email); |

출력

|  |
| --- |
| 홍길동 16 hong@skhu.net |

### dassign5.js

|  |
| --- |
| let person = { name: "홍길동", age: 16, email: "hong@skhu.net" };  let { name, age, email } = person;  console.log(name, age, email); |

출력

|  |
| --- |
| 홍길동 16 hong@skhu.net |

let name = person.name;

let age = person.age;

let email = person.email;

위 코드와 아래 코드는 완전히 같다.

let { name, age, email } = person;

let { name, age, email } = person;

name, age, email 변수가 생성되고,

person 객체가 쪼개져서(분해되서), 객체 내부 속성값들 name, age, email 변수에 대입된다.

이때 속성의 순서는 중요하지 않다.

변수 이름과 동일한 이름의 속성의 값이 대입된다.

let { age, email, name } = person;

위와 같인 변수 순서가 달라져도, 이름이 정확하기 때문에, 결과는 동일하다.

### dassign6.js

|  |
| --- |
| let person = { name: "홍길동", age: 16, email: "hong@skhu.net" };  let { nm, ag, em } = person;  console.log(nm, ag, em); |

출력

|  |
| --- |
| undefined undefined undefined |

let { nm, ag, em } = person;

변수 이름과 동일한 이름의 속성이 없으면 변수에 값이 대입되지 않는다.

### dassign7.js

|  |
| --- |
| let person = { name: "홍길동", age: 16, email: "hong@skhu.net" };  let { name: nm, age: ag, email: em } = person;  console.log(nm, ag, em); |

출력

|  |
| --- |
| 홍길동 16 hong@skhu.net |

let { name: nm, age: ag, email: em } = person;

변수 이름과 속성 이름이 일치하지 않으면,

위와 같이 속성명을 명시해 줄 수 있다.

### dassign8.js

|  |
| --- |
| let person = { name: "홍길동", age: 16, email: "hong@skhu.net" };  print1(person);  function print1(person) {     console.log(person.name, person.age, person.email);  }  print2(person);  function print2({ name, age, email }) {     console.log(name, age, email);  } |

파라미터 값 전달에서 구조 분해 할당

출력

|  |
| --- |
| 홍길동 16 hong@skhu.net  홍길동 16 hong@skhu.net |

# 전개 연산자 (spread operator)

## 배열 분해

### spread1.js

|  |
| --- |
| let a = [1, 2];  let b = [3, 4];  let c = [5, 6];  let m = [a, b, c];  console.log(m.length)  console.log(m);  let n = [...a, ...b, ...c];  console.log(n.length);  console.log(n); |

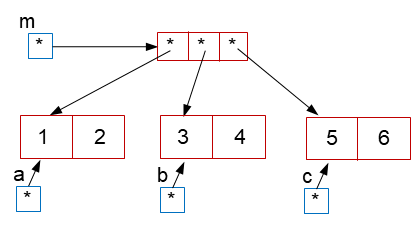
출력

|  |
| --- |
| 3  [ [ 1, 2 ], [ 3, 4 ], [ 5, 6 ] ]  6  [ 1, 2, 3, 4, 5, 6 ] |

let m = [a, b, c];

위 코드에서 생성되는 배열의 크기는 3 이다.

그리고 그 3 항목의 각각 a, b, c 배열이다.



let n = [...a, ...b, ...c];

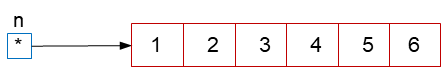
... 부분은 분해 연산자(spread operator) 이다.

...a 부분은 a 배열의 내용을 쪼개서 여기에 넣으라는 의미이다.

a, b, c 배열의 내용이 쪼개져서 n 배열이 생성된다.

따라서 위 코드는 아래 코드와 동일하다.

let n = [1, 2, 3, 4, 5, 6];



### spread2.js

|  |
| --- |
| function print(...params) {      console.log("\n파라미터 수:", params.length);      for (let i = 0; i < params.length; ++i)          console.log("  " + i + "번 파라미터: " + params[i]);  }  let a = ["one", "two", "three"];  print(a);  print(...a); |

출력

|  |
| --- |
| 파라미터 수: 1  0번 파라미터: one,two,three  파라미터 수: 3  0번 파라미터: one  1번 파라미터: two  2번 파라미터: three |

function print(...params) {

여기서 ...params 부분은 가변 파라미터 함수 문법.

전달된 파라미터 값들을 배열로 묶어서 params 변수에 그 배열을 대입해서 전달하라.

print( a );

전달되는 파라미터는 한 개의 배열이다..

따라서 위 코드는 아래 코드와 동일하다.

print( ["one", "two", "three"] );

전달되는 값 1개 ["one", "two", "three"] 를 배열로 묶어서 params에 대입.

params 변수에 전달되는 값은 [ ["one", "two", "three"] ]

print( ...a );

a 배열의 내용을 쪼개어서 파라미터로 전달한다.

따라서 전달되는 파라미터는 새 개의 문자열이다.

따라서 위 코드는 아래 코드와 동일하다.

print( "one", "two", "three" );

전달되는 값 3개 "one", "two", "three" 를 배열로 묶어서 params에 대입.

params 변수에 전달되는 값은 [ "one", "two", "three" ]

## 객체 분해

### spread3.js

|  |
| --- |
| let person1 = { name: "홍길동", age: 16 };  let person2 = { ...person1, email: "hong@skhu.net" };  console.log(person1);  console.log(person2); |

출력

|  |
| --- |
| { name: '홍길동', age: 16 }  { name: '홍길동', age: 16, email: 'hong@skhu.net' } |

let person2 = { ...person1, email: "hong@skhu.net" };

...person1 부분은 person1 객체의 속성들을 분해해서 여기에 넣으라는 의미이다.

따라서 위 코드는 아래 코드와 동일하다.

let person2 = { name: "홍길동", age: 16, email: "hong@skhu.net" };

# module

## 모듈(module)이란?

다른 javascript 소스코드 파일에 구현된 함수, 클래스, 객체 등을 가져와서 사용하기 위한 문법이다.

javascript 소스코드 파일에 구현된 함수/클래스/객체 등은

기본적으로 그 소스코드 파일 내부에서만 사용될 수 있다.

A.js 파일에 구현된 함수/클래스/객체 등을 B.js 파일에서 사용할 수 있으려면,

A.js 에서 제공(export)해야 하고 B.js 에서 수입(import) 해야 한다.

## package.json 파일 수정

|  |
| --- |
| {    "name": "node1",    "version": "1.0.0",    "description": "",    "main": "index.js",    "scripts": {      "test": "echo \"Error: no test specified\" && exit 1"    },    "keywords": [],    "author": "",    "license": "ISC",    "devDependencies": {      "eslint": "^8.35.0"    },    "type": "module"  } |

ES6 module 문법을 사용하려면, package.json 파일에 노란색으로 칠한 부분이 필요하다.

## export & import

### math1.js

|  |
| --- |
| function sum(...numbers) {      let result = 0;      for (let i of numbers)          result += i;      return result;  }  function average(...numbers) {      return sum(...numbers) / numbers.length;  }  export { average, sum }; |

export { average, sum };

average, sum 함수를 export 한다.

즉 math1.js 모듈은 average, sum 함수를 제공(export) 한다.

### test\_math1a.js

|  |
| --- |
| import { average, sum } from './math1.js'  console.log(average(3, 4, 5, 6));  console.log(sum(3, 4, 5, 6)); |

출력

|  |
| --- |
| 4.5  18 |

import { average, sum } from './math1.js'

math1js 모듈에서 export하는 average, sum 함수를 수입(import) 한다.

math1.js 모듈에 구현된 average, sum 함수를 다른 곳에서 사용하려면

위와 같이 export, import 해야 한다.

### test\_math1b.js

|  |
| --- |
| import { average } from './math1.js'  console.log(average(3, 4, 5, 6)); |

math1.js모듈이 export 하는 것을 전부 import 해야 하는 것은 아니고,

필요한 것만 import 해도 된다.

test\_math1b.js 소스코드에서 average 함수만 사용하기 때문에

average 함수만 import 했다.

다음과 같이 하나씩 따로 export 해도 된다.

|  |
| --- |
| export function sum(...numbers) {    let result = 0;    for (let i of numbers)        result += i;    return result;  }  export function average(...numbers) {    return sum(...numbers) / numbers.length;  } |

|  |
| --- |
| import { average, sum } from './math1.js'  console.log(average(3, 4, 5, 6));  console.log(sum(3, 4, 5, 6)); |

import는 동일하게...

## export default

### math2.js

|  |
| --- |
| let PI = 3.141592;  function sum(...numbers) {      let result = 0;      for (let i of numbers)          result += i;      return result;  }  function average(...numbers) {      return sum(...numbers) / numbers.length;  }  export { average, sum, PI };  export default { average, sum, PI }; |

### test\_math2a.js

|  |
| --- |
| import { average, PI } from './math2.js'  console.log(average(3, 4, 5, 6));  console.log(PI); |

출력

|  |
| --- |
| 4.5  3.141592 |

### test\_math2b.js

|  |
| --- |
| import Math from './math2.js'  console.log(Math.average(3, 4, 5, 6));  console.log(Math.PI); |

export default 된 것은 위와 같은 변수 한 개로 import 할 수 있다.

출력

|  |
| --- |
| 4.5  3.141592 |

export { average, sum, PI };

위 코드로 export 한 것은 다음과 같이 import 한다.

import { average, PI } from './math2.js'

export default { average, sum, PI };

위 코드로 export 한 것은 다음과 같이 import 한다.

import Math from './math2.js'

### test\_math2c.js

|  |
| --- |
| import Math, { average, PI } from './math2.js'  console.log(average(3, 4, 5, 6));  console.log(PI);  console.log(Math.average(3, 4, 5, 6));  console.log(Math.PI); |

출력

|  |
| --- |
| 4.5  3.141592  4.5  3.141592 |

두 import 방식을 혼합해서 사용해도 된다.

## 하나씩 export

### math3.js

|  |
| --- |
| export let PI = 3.141592;  export function sum(...numbers) {      let result = 0;      for (let i of numbers)          result += i;      return result;  }  export function average(...numbers) {      return sum(...numbers) / numbers.length;  } |

위와 같이 하나씩 export 할 수 있다.

### test\_math3a.js

|  |
| --- |
| import { average, PI } from './math2.js'  console.log(average(3, 4, 5, 6));  console.log(PI); |

### test\_math3b.js

|  |
| --- |
| import \* as Math from './math2.js'  console.log(Math.average(3, 4, 5, 6));  console.log(Math.PI); |

이렇게 import 할 수도 있다.

# 함수와 화살표 함수

## 메소드와 this

### func1.js

|  |
| --- |
| let rectangle1 = {      width: 4,      height: 5,      method1: function() {          let area = this.width \* this.height;          console.log(area);      }  };  rectangle1.method1(); |

출력

|  |
| --- |
| 20 |

rectangle1 객체의 width 속성값은 4 이고 값의 타입은 number 이다.

rectangle1 객체의 method1 속성값의 타입은 function 이다.

javascript 문법에서 함수도 자료형(data type)이다.

Java 언어에서 메소드는 int나 String 같은 자료(data)가 아니다.

Java 언어에서 함수나 메소드는 number나 string 같은 자료(data)처럼 사용할 수 있다.

숫자나 문자열 데이터를 변수에 대입하고, 파라미터 값으로 전달하고, 리턴값으로 리턴할 수 있는 것 처럼,

함수를 변수에 대입하고, 파라미터 값으로 전달하고, 리턴값으로 리턴할 수 있다.

rectangle1 객체의 method1 속성을 메소드라고 생각할 수도 있다.

### func2.js

|  |
| --- |
| let rectangle1 = {      width: 4,      height: 5,      method1() {          let area = this.width \* this.height;          console.log(area);      }  };  rectangle1.method1(); |

출력

|  |
| --- |
| 20 |

func1.js와 func2.js는 동일한 코드이다.

javascipt 함수/메소드 본문의 this는, 그 함수/메소드가 호출되는 시점의 this 객체이다.

즉 아래 코드로 method1을 호출하면, method1 본문의 this는 rectangle1 객체를 가르킨다.

rectangle1.method1();

### func3.js

|  |
| --- |
| let rectangle1 = {      width: 4,      height: 5,      method1() {          let area = this.width \* this.height;          console.log(area);      }  };  let method1 = rectangle1.method1;  method1(); |

출력

|  |
| --- |
| 에러 |

javascript 메소드 본문의 this가 무엇을 가르키는 지는, 그 메소드를 호출하는 방법에 따라 다르다.

예를 들어 어래 코드로 호출하면, method1 본문의 this는 rectangle1 객체를 가르키지 않는다.

let method1 = rectangle1.method1;

method1 변수가 만들어지고, 이 변수에 rectangle1.method1 값이 대입된다.

대입되는 값은 함수이다.

method1();

method1 변수에 대입된 함수를 호출한다.

###.method1() 형태로 호출하지 않았기 때문에, 이 함수 본문의 this는 어떤 객체를 가르키지 않는다.

## inner 함수

### func4.js

|  |
| --- |
| let rectangle1 = {      width: 4,      height: 5,      method1() {          let getArea = function() {              return this.width \* this.height;          };          console.log(getArea());      }  };  rectangle1.method1(); |

출력

|  |
| --- |
| 에러 |

getArea는 method1 내부의 지역 변수이다.

method1 변수에 대입된 값은 함수이다.

### func5.js

|  |
| --- |
| let rectangle1 = {      width: 4,      height: 5,      method1() {          function getArea() {              return this.width \* this.height;          };          console.log(getArea());      }  };  rectangle1.method1(); |

출력

|  |
| --- |
| 에러 |

func4.js와 func5.js는 동일한 코드이다.

getArea 함수는 method1 내부에 정의되었다.

다른 함수(메소드) 내부에 정의된 함수를 inner 함수라고 부른다.

javascipt 함수/메소드 본문의 this는, 그 함수/메소드가 호출되는 시점의 this 객체이다.

즉 아래 코드로 getArea를 호출하면, getArea 본문의 this는 어떤 객체를 가르키지 않는다.

getArea()

###.getArea() 이렇게 호출하지 않았으므로, getArea 본문의 this는 어떤 객체를 가르키지 않는다.

## 화살표 함수

### func6.js

|  |
| --- |
| let rectangle1 = {      width: 4,      height: 5,      method1() {          let getArea = () => {              return this.width \* this.height;          };          console.log(getArea());      }  };  rectangle1.method1(); |

출력

|  |
| --- |
| 20 |

getArea는 method1 내부의 지역 변수이다.

method1 변수에 대입된 값은 함수이고, 이 함수는 화살표 함수(arrow function) 이다.

### 화살표 함수의 예

|  |
| --- |
| function add(a, b) {      return a + b;  } |

위 함수를 아래처럼 구현할 수도 있다.

|  |
| --- |
| let add = (a, b) => {      return a + b;  } |

화살표 함수 본문이 리턴문 한 개 뿐이면, 다음과 같이 좀더 간결하게 구현할 수도 있다.

|  |
| --- |
| let add = (a, b) => a + b; |

### func7.js

|  |
| --- |
| let rectangle1 = {      width: 4,      height: 5,      method1() {          let getArea = () => this.width \* this.height;          console.log(getArea());      }  };  rectangle1.method1(); |

출력

|  |
| --- |
| 20 |

###.함수() 형태로 호출하지 않아도, 화살표 함수의 본문의 this는 어떤 객체를 가르킬 수 있다.

화살표 함수 본문의 this는, 그 함수가 포함된 메소드의 this와 같다.

rectangle1.method1();

위 코드에 의해서 method1이 호출되었으므로,

method1 본문의 this는 rectangle1 객체를 가르킨다.

이 method1 본문에 들어있는 화살표 함수의 this도 rectangle1 객체를 가르킨다.

### func8.js - bind

|  |
| --- |
| let rectangle1 = {      width: 4,      height: 5,      method1() {          let getArea = function() {              return this.width \* this.height;          };          let getArea2 = getArea.bind(rectangle1);          console.log(getArea2());      }  };  rectangle1.method1(); |

let getArea2 = getArea.bind(rectangle1);

getArea 함수를 복사해서 새 함수를 만들고,

그 새 함수를 getArea2 지역 변수에 대입한다.

그 새 함수 본문의 this는 rectangle1 객체를 가르킨다.

getArea2()

###.getArea2() 형태로 호출하지 않았지만,

getArea2 본문의 this는 이미 rectangle1 객체를 가르키도록 설정되어있다.

## 요약

### function 키워드 문법 함수

function 키워드 문법으로 만든 함수 본문의 this는 무엇을 가르키는가?

let func2 = func1.bind(obj)

위와 같은 코드로 obj 객체에 bind된 func2 본문의 this는 obj 객체를 가르킨다.

obj.func(...)

위와 같은 코드로 호출된 함수 본문의 this는 obj 객체를 가르킨다.

func(...)

위와 같은 코드로 호출된 함수가 bind 되지도 않았다면,

이 함수 본문의 this는 어떤 객체를 가르키지 않는다.

### 화살표 함수

화살표 함수 본문의 this는, 화살표 함수 소스코드를 둘러 싸고 있는 객체를 가르킨다

# 배열의 메소드

|  |
| --- |
| const a1 = new Array(5);  a1.fill(0);  const a2 = a1.map((value, index) => index \* 10);  console.log(a1, a2);  export {} |

# typescript 기초

## typescript 설치

### typescript 트랜스파일러

javascript 인터프리터는 typescript 소스코드를 실행하지 못한다.

typescript 소스코드를 javascript 소스코드로 변환해서 실행해야 한다.

typescript 소스코드를 javascript 소스코드로 변환하는 도구가 typescript 트랜스파일러이다.

### ts-node

node.js는 javascript 소스코드만 실행할 수 있다.

typescript 소스코드를 실행하려면 ts-node가 필요하다.

### 설치

다음 명령을 실행하여 typescript 트랜스파일러와 ts-node를 설치하자.

|  |
| --- |
| npm install -g typescript ts-node |

다음 명령을 실행하여 설치를 확인하자.

|  |
| --- |
| tsc --version |

|  |
| --- |
| ts-node --version |

## 프로젝트 생성

### 프로젝트 디렉토리 생성

적당한 위치에 typescript 프로젝트 디렉토리를 만들자.

예: C:/PJ/frontend/typescript1

### Visual Studio Code

Visual Studio Code에서 그 디렉토리를 열자.

메뉴: File - Open Folder

### package.json 파일 생성

package.json 파일은 node.js 프로젝트의 기본 설정파일이다.

프로젝트 디렉토리에서 다음 명령을 실행하여 package.json 파일을 생성하자.

|  |
| --- |
| npm init -y |

### tsconfig.json 파일 생성

tsconfig.json 파일은 typescript 트랜스파일러 설정 파일이다.

다음 명령을 실행하여 tsconfig.json 파일을 생성하자.

|  |
| --- |
| tsc --init |

### .eslintrc 설정 파일 생성

다음 명령을 실행하여 .eslintrc 설정 파일을 생성하자.

|  |
| --- |
| npm init @eslint/config |

|  |
| --- |
| Text  Description automatically generated |

위 항목을 선택하고 엔터

|  |
| --- |
| Text  Description automatically generated |

위 항목을 선택하고 엔터

|  |
| --- |
| A picture containing logo  Description automatically generated |

위 항목을 선택하고 엔터

|  |
| --- |
|  |

Yes를 선택하고 엔터

|  |
| --- |
|  |

위 항목을 선택하고 엔터

|  |
| --- |
| A picture containing logo  Description automatically generated |

위 항목을 선택하고 엔터

|  |
| --- |
|  |

Yes를 선택하고 엔터

|  |
| --- |
| A picture containing text  Description automatically generated |

위 항목을 선택하고 엔터

## 변수 선언

### test1.ts

|  |
| --- |
| let a : number;  let s : string;  let b : boolean;  a = 3;  s = "hello";  b = true;  console.log(a);  console.log(s);  console.log(b);  export {} |

다음 명령을 실행하면 프로젝트 디렉토리 아래 모든 \*.ts 파일들이 \*.js 파일로 트랜스 파일된다.

|  |
| --- |
| tsc |

\*.ts 파일들이 \*.js 파일로 트랜스 파일하고, \*.js 파일을 node로 실행할 수도 있지만,

\*.ts 파일을 ts-node로 바로 실행할 수도 잇다.

\*.

test1.ts 파일을 실행하는 명령은 다음과 같다.

|  |
| --- |
| ts-node test1.ts |

let a : number;

let s : string;

let b : boolean;

typescript 에서는 위와 같이 변수 타입이 필요하다.

a 변수에는 숫자만 대입할 수 있고, s 변수에는 문자열만 대입할 수 있고,

b 변수에는 true/false 값만 대입할 수 있다.

export {}

이 모듈은 아무것도 export 하지 않는다는 선언이다.

이 선언이 필요한 이유는 일단 생략.

### test2.ts

|  |
| --- |
| let a : number;  a = 3;  console.log(a);  a = "hello";  console.log(a);  export {} |

a 변수는 number 타입이라고 선언되었기 때문에 문자열을 대입할 수 없다.

트랜스 파일 단계에서 에러가 발생한다.

|  |
| --- |
|  |

## 타입의 종류

|  |  |
| --- | --- |
| number | 숫자 |
| string | 문자열 |
| boolean | true/false |
| any | 어떤 타입의 값이나 다 허용하는 타입. 이 타입을 사용하는 것은 바람직하지 않다. |
| null | null은 값이면서 타입이기도 하다 |
| undefined | undefined는 값이면서 타입이기도 하다 |
| void | 리턴 값이 없는 함수의 타입 |

## 배열

### array1.ts

|  |
| --- |
| const a1 : number[] = [1, 2, 3, 4];  const a2 : string[] = ["one", "two", "three"];  console.log(a1);  console.log(a2);  export {} |

### array2.ts

|  |
| --- |
| const a1 : number[][] = [[1, 2], [3, 4], [5, 6]];  const a2 : string[][] = [["one", "two"], ["three", "four"]];  console.log(a1);  console.log(a2);  export {} |

2차원 배열

## 함수 선언

### function1.ts

|  |
| --- |
| function add(a : number, b : number) : number {      return a + b;  }  const result : number = add(3, 4);  console.log(result);  export {} |

function add(a : number, b : number) : number {

add 함수의 파라미터 a, b의 타입은 number

add 함수의 리턴 타입도 number 이다.

const result : number = add(3, 4);

result 의 타입도 number 이다.

### function2.ts

|  |
| --- |
| const add = (a: number, b: number) : number => {      return a + b;  }  const result: number = add(3, 4);  console.log(result);  export {} |

add 함수를 화살표 함수 문법으로 구현

### function3.ts

|  |
| --- |
| const add = (a: number, b: number) : number => a + b;  const result: number = add(3, 4);  console.log(result);  export {} |

add 함수를 화살표 함수 문법으로 구현

## 타입 별칭 (type alias)

### alias1.ts

|  |
| --- |
| type Person = { name : string, age: number };  const p1 : Person = { name: "홍길동", age: 16 };  const p2 : Person = { name: "임꺽정", age: 19 };  console.log(p1);  console.log(p2);  export {} |

type Person = { name : string, age: number };

Person 타입을 선언한다.

Person 타입은 객체이고 이 객체에는 name 속성과 age 속성이 있다.

const p1 : Person = { name: "홍길동", age: 16 };

p1의 타입은 Person 이고 값은 { name: "홍길동", age: 16 } 이다.

### alias2.ts

|  |
| --- |
| type Person = { name : string, age: number };  const p1 : Person = { name: "홍길동" };  const p2 : Person = { name: "임꺽정", age: 19, email: "lim@skhu.net" };  console.log(p1);  console.log(p2);  export {} |

const p1 : Person = { name: "홍길동" };

age 속성값을 부여하지 않았으므로 컴파일 에러가 발생한다.

const p2 : Person = { name: "임꺽정", age: 19, email: "lim@skhu.net" };

Person 타입에 포함되지 않은 email 속성값을 부여했으므로 컴파일 에러가 발생한다.

### alias3.ts - optional property

|  |
| --- |
| type Person = { name : string, age?: number, email?: string };  const p1 : Person = { name: "홍길동" };  const p2 : Person = { name: "임꺽정", age: 19, email: "lim@skhu.net" };  console.log(p1);  console.log(p2);  export {} |

type Person = { name : string, age?: number, email?: string };

속성 이름에 ? 기호가 붙으면, 생략 가능한 속성이 된다.

const p1 : Person = { name: "홍길동" };

age, email 속성값을 부여하지 않았지만,

이 두 속성이 생략 가능한 속성이기 때문에 에러가 발생하지 않는다.

### alias4.ts - readonly property

|  |
| --- |
| type Person = { readonly name : string, age: number, email: string };  const p1 : Person = { name: "홍길동", age: 16, email: "hong@skhu.net" };  const p2 : Person = { name: "임꺽정", age: 19, email: "lim@skhu.net" };  p1.age = 17;  p1.name = "Hone, Gil-dong"; // 컴파일 에러  console.log(p1);  console.log(p2);  export {} |

속성 이름 앞에 readonly 키워드를 붙이면,

그 속성은 객체를 처음 생성할 때만 값을 지정할 수 있고, 그 이후로는 값을 변경할 수 없다.

따라서 다음 줄에서 컴파일 에러가 발생한다.

p1.name = "Hone, Gil-dong"; // 컴파일 에러

### alias5.ts - 배열

|  |
| --- |
| type Person = { readonly name : string, age: number, email: string };  const a : Person[] = [ { name: "홍길동", age: 16, email: "hong@skhu.net" },                         { name: "임꺽정", age: 19, email: "lim@skhu.net" } ];  for (let person of a)      console.log(person);  export {} |

## 열거형

### enum1.ts

|  |
| --- |
| enum Color { BLACK, RED, GREEN, BLUE, WHITE }  const color1 : Color = Color.RED;  const color2 : Color = Color.BLUE;  console.log(color1);  console.log(color2);  export {} |

출력

|  |
| --- |
| 1  3 |

Java 언어의 enum 문법과 매우 유사하다.

여러 개의 상수를 선언하는 문법이다.

기본적으로 상수에 0부터 시작되는 숫자형 값이 부여된다.

enum Color { BLACK, RED, GREEN, BLUE, WHITE }

BLACK 0

RED 1

GREEN 2

BLUE 3

WHITE 4

### enum2.ts

|  |
| --- |
| enum Color { BLACK = 10, RED = 11, GREEN = 12, BLUE = 13, WHITE = 14 }  const color1 : Color = Color.RED;  const color2 : Color = Color.BLUE;  console.log(color1);  console.log(color2);  export {} |

출력

|  |
| --- |
| 11  13 |

위와 같이 상수 값을 명시적으로 적어줄 수도 있다.

enum Color { BLACK = 10, RED = 11, GREEN = 12, BLUE = 13, WHITE = 14 }

### enum3.ts

|  |
| --- |
| enum Color { BLACK = "BLACK", RED = "RED", GREEN = "GREEN", BLUE = "BLUE", WHITE = "WHITE" }  const color1 : Color = Color.RED;  const color2 : Color = Color.BLUE;  console.log(color1);  console.log(color2);  export {} |

출력

|  |
| --- |
| RAD  BLUE |

위와 같이 상수에 문자열 값을 적어줄 수도 있다.

enum Color { BLACK = "BLACK", RED = "RED", GREEN = "GREEN", BLUE = "BLUE", WHITE = "WHITE" }