

## 06

프론트 웹 개발이 애플리케이션이라는 큰 단위로 발전하게 됨에 따라 두 가지 변화가 일어났다.

### (1) 자바스크립트 프레임 워크의 등장

- 코드가 많아지다보니, 프레임 워크를 쓰지 않고 개발하기가 거의 불가능하게 되었다.

### (2) TypeScript

- 자바스크립트 언어의 한계를 극복하기 위한 언어.

- ES5 시절에 나왔다. 그런데 그 이후에 타입 스크립트를 많이 참조하여 ES6가 나왔다.

- 그래서 ES6가 지원하는 부분이랑

TypeScript가 지원하는 부분이 겹칠 수 있다.

- 하지만, TypeScript가 지원하는 부분이 훨씬 더 많다.

자신들은 데이터 type을 지원하는 언어라는 것을 부각시키기 위해서 이렇게 명명

TypeScript

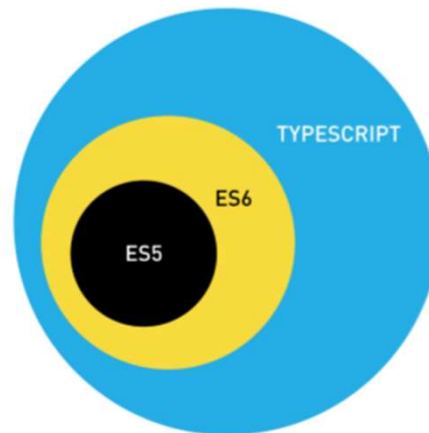
우리는 ES6 이후에 나온 js 기능과 TypeScript가 겹치지 않는 부분만 살펴보겠다.  
즉, JavaScript가 지원하지 않는 기능만 보겠다.

# TypeScript 란

- 타입스크립트 (TypeScript)는 MS에서 만든 대규모 어플리케이션 개발을 위한 소프트웨어 언어
- ES6 + 정적 타입
- ES6 는 Supset, Typescript 는 ES6 의 Super set
- 타입스크립트는 ES6 의 모든 기능을 지원한다.
- 정적 타입을 지원함으로 코드의 오류를 줄이고 쉽게 디버깅이 된다.

JS는 한번 할당한 변수의 타입을 재 할당을 통해 변경할 수 있는데, 타입 스크립트는 그것을 방지한다.

문법 실수를 알아서 체크해준다는 것은 개발자의 실수를 미연에 방지해주기에, 프로그램이 더욱 견고하게 작성될 수 있다.



# 개발환경

---

- 타입스크립트를 브라우저의 JS Engine 이 해석하지 못한다.
- 타입스크립트 코드를 브라우저에서 해석 가능한 자바스크립트 코드로 변형시켜야 한다.
- 컴파일러(트랜스파일러) – tsc      **컴파일 : 사람이 만든 소스코드를 기계가 이해할 수 있는 기계어로 바꾸는 과정**

>npm install typescript

>npx tsc main.ts

**개발은 타입스크립트로 하고, 파일을 js로 바꿔줘야 하기 때문에  
트랜스파일 과정이 필요하다. (그냥 컴파일이라고 부르기도 한다)**

# 개발환경


---

- tsconfig.json 활용

> npm init

> npm install -D typescript

> npx tsc -init

- 타입스크립트 초기 설정 파일을 만들어 준다. (tsconfig.json)
- 

- <https://typescript-kr.github.io/pages/tsconfig.json.html>

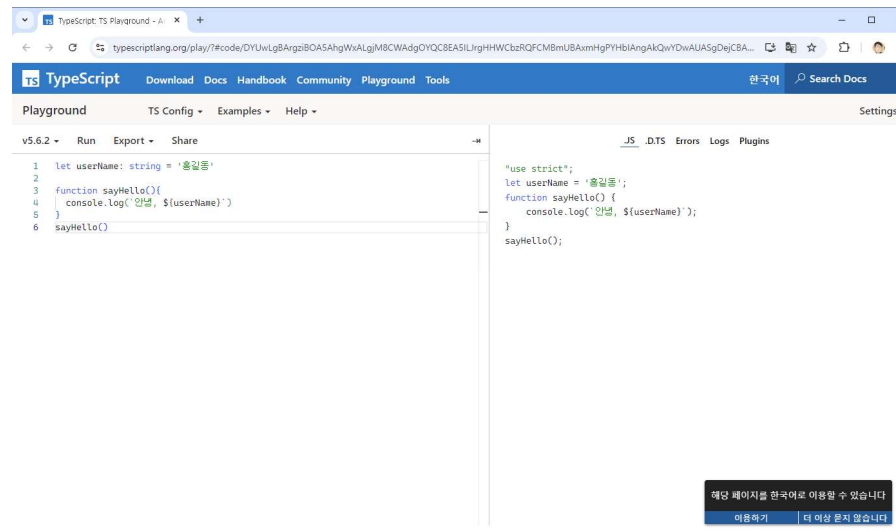
개발자가 tsconfig.json에 설정한대로 tsc가 움직인다.

```
{
  "compilerOptions": {
    "outDir": "./build/",
    "allowJs": true,
    "esModuleInterop": true,
    "resolveJsonModule": true,
    "sourceMap": true,
    "noImplicitAny": true,

    "target": "es5",
    "jsx": "react",
    "module": "commonjs"
  },
  "include": [".src/**/*.ts"],
  "exclude": ["node_modules", "**/*.spec.ts"]
}
```

# 개발환경

- 타입스크립트 플레이 그라운드
  - 브라우저에서 간단하게 컴파일 하는 툴
  - <https://www.typescriptlang.org/play>



# 타입

---

let a: number = 10

- number, string, boolean, any, array, null, undefined, void, union, never, object, 사용자 지정 타입

```
let fullName: string = 'Bob Bobbington';  
let age: number = 37;  
let sentence: string = `Hello, ${fullName}`  
console.log(sentence);  
age = age+1  
//fullName=10//error
```

# 타입

---

- any : 모든 타입의 데이터. 이 타입이면 타입 검사를 수행하지 않음

```
let notSure: any = 4;  
notSure = "maybe a string instead";  
notSure = false;
```

- array :

```
let a: number[] = [10, 20]  
let b: Array<number> = [1,2]
```

# 타입

---

- void : 결과값 리턴하지 않는 함수
- 함수의 리턴타입으로 사용하여 함수의 리턴값이 없음을 명시
- 변수의 타입으로 선언 가능하지만 undefined 값만 할당 가능함으로 의미 없다.

```
const b = (): void => {
```

```
}
```

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



# 타입

---

- Union 은 여러타입을 허용하고자 할 때

```
let c: number|string;  
c=10  
c="hello"
```

- 사용자지정 타입
- 복잡한 타입을 지정할 때 사용

```
let obj: {id: number, name: string}  
obj = {  
  id: 10,  
  name: "hello"  
}
```

# 타입

---

- null 은 값일수도 있고 타입일 수도 있다.
- undefined 도 값일 수도 있고 타입일 수도 있다. 할당된 값이 없다는 개념
- --strictNullChecks 모드에서 유용

```
var data1: null = null
var data2: null = 10//error
var data3: undefined = undefined
var data4: undefined = 10//error
```

```
var data5: number = null//tsc main.ts (ok), tsc main.ts --strictNullChecks (error)
var data6: number | null = null//tsc main.ts (ok), tsc main.ts --strictNullChecks (ok)
```

# 타입

---

- never
- 함수의 리턴타입으로 사용하여 종료되지 않거나 의미있는 결과 값이 리턴되지 않는다는 것을 명시적으로 선언

```
function error(message: string): never {  
    throw new Error(message);  
}
```

```
// Inferred return type is never  
function fail() {  
    return Error("Something failed");  
}
```

```
function infiniteLoop(): never {  
    while (true) {  
    }  
}
```

# 타입

---

- object
- 데이터가 boolean, string, number, null, undefined 가 아님을 선언

```
function create(o: object | null): void {}
```

```
create({ prop: 0 }); // OK  
create(null); // OK
```

```
create(42); // Error  
create("string"); // Error  
create(false); // Error  
create(undefined); // Error
```

# 타입

---

- 제네릭 제공한다.
- 제네릭으로 형식타입을 선언하고 실제 이용시에 실제 타입을 지정해서 사용할 수 있다.

```
function a<T>(arg1: T[], arg2: T[]): T[] {  
    return arg1.concat(arg2)  
}  
let b = a<number>([10, 20], [30, 40])
```

- Typealias
- Type 예약어로 선언한다.

```
type MyType2 = {id: number, name: string}  
let b: MyType2 = {id: 10, name: "hello"}
```

# 타입

---

- optional
- 생략 가능한 데이터의 타입 표현

```
let c: {id: number, name?: string}
c = {id: 20}
c = {id: 30, name: "hello"}
```

- readonly
- 읽기 전용으로 사용해야 하는 데이터 표현

```
type MyType3 = {id: number, name?: string, readonly email: string}
```

# 타입

---

- tuple
- 튜플 타입은 배열 내부의 타입과 순서를 일치시키기 위해서 사용

```
let d:[number, string]  
d = [10, "hello"]
```

- Intersection Type
- 두 타입을 모두 만족해야 한다.
- & 기호로 한다.
- 공통뿐만 아니라 공통이지 않은 부분도 모두 포함되어야 한다.

```
type PersonTypeInter = PersonTypeA & PersonTypeB;
```

# Enum

---

- Enum(열거형)
  - 상수에 기본적으로 index 값이 지정
  - 원한다면 값 할당(문자, 숫자) 지정 가능
  - 숫자로 값을 지정하면 열거 상수 중 값 할당이 안되어도 된다. 그렇게 되면 마지막 지정한 값에 +1 해서 그 다음 값이 지정된다.

```
enum Media2 {  
    Newspaper = "신문",  
    Broadcasting = "방송",  
    SNS = "SNS",  
    Magazine = "잡지",  
    Youtube = "유튜브",  
}
```

```
let media2: Media2 = Media2.Youtube;
```



# Interface

---

- Interface

- 인터페이스는 interface 로 선언되는 타입 정의 기법

```
interface IEmp {  
  no: number;  
  name: string;  
  salary: number;  
}
```

- 이름이 동일한 인터페이스를 여러 개 선언해도 되며 중복되는 내용은 결합시켜 버린다.

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

```
interface IPerson { name: string; tel: string; }
```

```
let p5: IPerson = { name: "홍길동", tel: "010-111-2222", age: 20 };
```

# Interface

---

- Interface, 타입으로 사용

```
interface soemInterface2 {  
    label: string;  
}  
  
function some2(arg: soemInterface2) {  
    console.log(arg.label);  
}  
some2({label: "Size 10 Object"}); //ok  
some2({size: 10, label: "Size 10 Object"}); //error
```

# Interface

---

- Interface 상속
  - 인터페이스를 이용해 다른 인터페이스 타입을 만들 수 있다.
  - extends 를 이용한다.

```
interface IPerson2 { name: string; age: number; }
```

```
interface IEmployee extends IPerson2 { employeeId: string; dept: string; }
```

```
let e1: IEmployee = { employeeId: "E001", dept: "회계팀", name: "홍길동", age: 20 };
```

# Interface

---

- 함수타입을 만들 때 사용

```
interface MyInterface {  
    (arg1: string, arg2: string): boolean;  
}  
let myFun: MyInterface;  
myFun = function(src: string, sub: string): boolean {  
    let result = src.search(sub);  
    return result > -1;  
}
```

# Interface

---

- 클래스 타입을 만들 때 유용
- 클래스의 property 와 function 을 선언하고 어떤 클래스가 그 property 와 function 이 선언되어야 함을 강제

```
interface MyClassInterface {  
    data: boolean;  
    some(): boolean;  
}  
class MyClass implements MyClassInterface {} //error  
class MyClass2 implements MyClassInterface {  
    //ok...  
    data: boolean = true;  
    some(): boolean {  
        return true;  
    }  
}
```

# Access Modifier

---

- public, private, and protected
- default modifier는 public
- 클래스의 멤버 접근 범위 제한

```
class Super {  
    name: string;  
    protected email: string;  
    private address: string  
}  
class Sub extends Super {  
    some(){  
        this.name="kkang"  
        this.email="a@a.com"  
        this.address="seoul"//error  
    }  
}  
let superObj = new Super()  
superObj.name="kim"  
superObj.email="b@b.com"//error  
superObj.address="busan"//error
```

# Access Modifier

---

- 생성자 argument 에 접근제한자 추가하면 object member

```
class MyClass2 {  
    constructor(name: string, private email: string, public address: string){ }  
    some(){  
        // this.name='kkang'//error  
        this.email="a@a.com"  
        this.address="seoul"  
    }  
}  
let obj4=new MyClass2('kim','b@b.com','busan')  
obj4.name='lee'//error  
obj4.email='c@c.com'//error  
obj4.address='kangdong'
```

# abstract

---

- abstract 예약어로 property, method, class 를 추상형으로 정의
- 추상 클래스는 객체생성 불가

```
abstract class AbstractClass {  
    abstract name: string  
    abstract some()  
}  
class AbstractSubClass extends AbstractClass {  
    name: string  
    some(){  
  
    }  
}  
let obj2=new AbstractClass();//error  
let obj3=new AbstractSubClass();//ok
```





# 감사합니다

단단히 마음먹고 떠난 사람은  
산꼭대기에 도착할 수 있다.  
산은 올라가는 사람에게만 정복된다.



윌리엄 셰익스피어  
William Shakespeare