TypeScript의 소개와 개발 환경 구축

Introduction

자바스크립트는 1995년 넷스케이프사의 브렌던 아이크(Brendan Eich)가 자사의 웹브라우저인 Navigator 2에 탑재하기 위해 개발한 스크립트 언어이다. 초창기 자바스크립트는 웹 페이지의 보조적인 기능을 수행하기 위해 **한정적인 용도**로 사용되었다. 이 시기에 대부분 로직은 주로 웹서버에서 실행되었고 브라우저(클라이언트)는 서버로부터 전달받은 HTML과 CSS를 렌더링하는 수준이었다.

HTML5가 등장하기 이전까지 웹 애플리케이션은 플래시, 실버라이트, 액티브엑스와 같은 플러그인에 의존하여 인터랙티브한 웹페이지를 구축해왔다. 그러다가 HTML5가 등장함으로써 **플러그인**에 의존하던 구축 방식은 자바스크립트로 대체되었다. 또한 AJAX의 활성화로 데스크탑 애플리케이션과 유사한 사용자 경험을 제공할 수 있는 **SPA(Single Page Application)**가 대세가 되었다. 이로써 과거 서버 측이 담당하던 업무의 많은 부분이 클라이언트 측으로 이동하게 되었고, 자바스크립트는 웹의 어셈블리 언어로 불릴 만큼 중요한 언어로 그 위상이 높아지게 되었다.

모든 프로그래밍 언어에 장단점이 있듯이 자바스크립트도 언어가 잘 정제되기 이전에 서둘러 출시된 문제와 과거 웹페이지의 보조적인 기능을 수행하기 위해 한정적인 용도로 만들어진 **태생적 한계**로 좋은 점도, 나쁜 점도 많은 것이 사실이다.

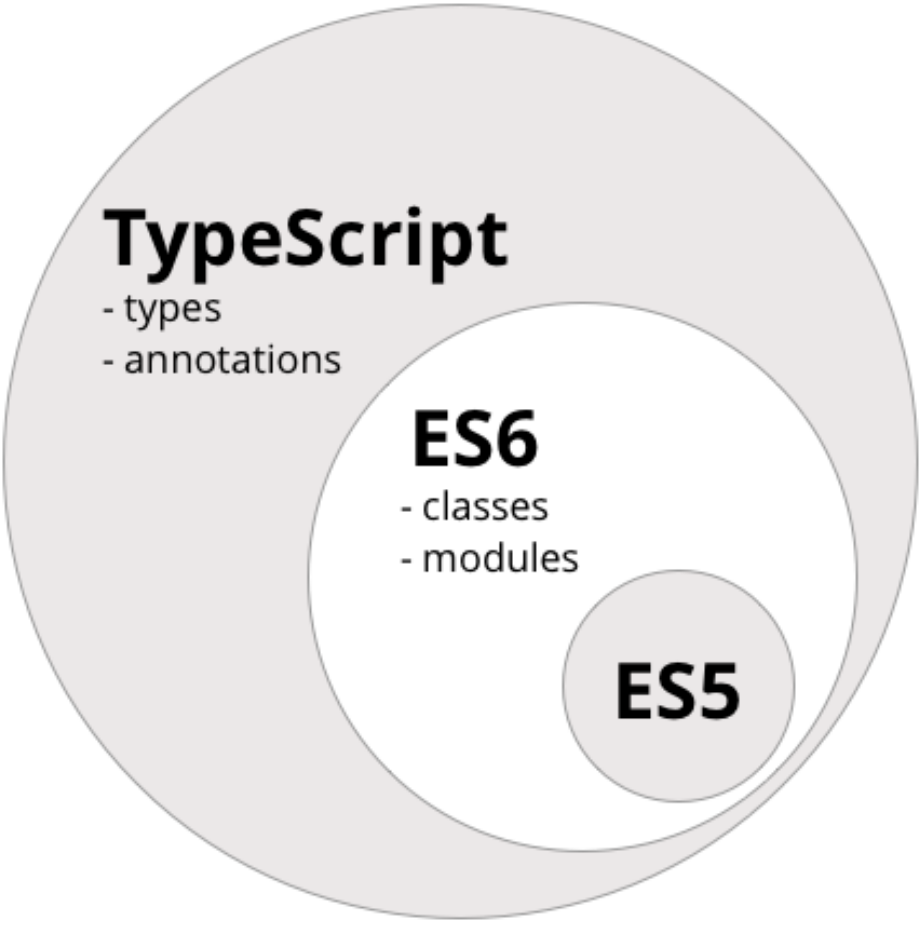
자바스크립트는 C나 Java와 같은 C-family 언어와는 구별되는 아래와 같은 특성이 있다.

- Prototype-based Object Oriented Language
- Scope와 this
- 동적 타입(dynamic typed) 언어 혹은 느슨한 타입(loosely typed) 언어

이와 같은 특성은 클래스 기반 객체지향 언어(Java, C++, C# 등)에 익숙한 개발자를 혼란스럽게 하며 코드가 복잡해질 수 있고 디버그와 테스트 공수가 증가하는 등의 문제를 일으킬 수 있어 특히 규모가 큰 프로젝트에서는 주의하여야 한다.

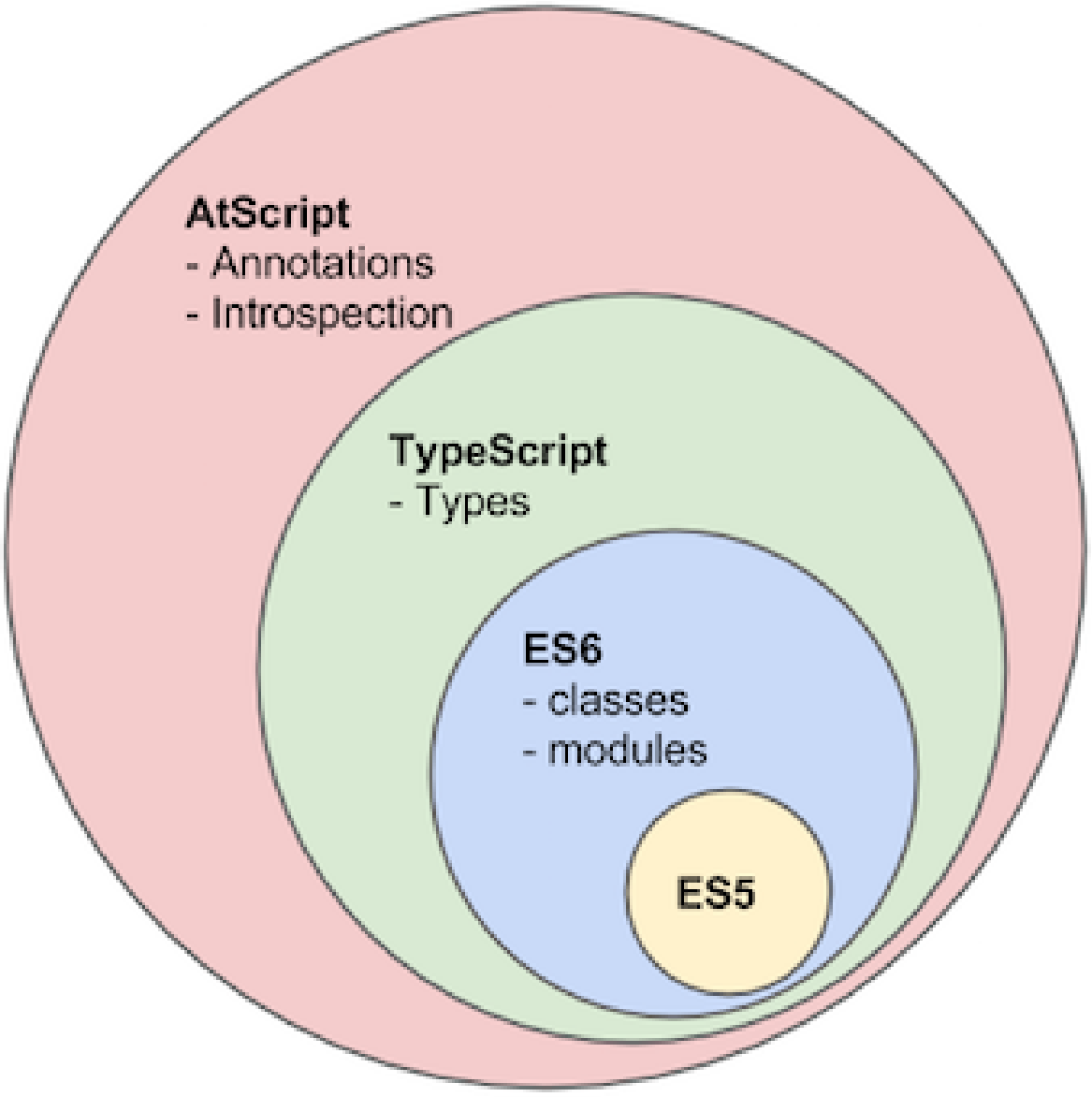
이같은 자바스크립트의 태생적 문제를 극복하고자 CoffeeScript, Dart, Haxe와 같은 **AltJS**(자바스크립트의 대체 언어)가 등장하였다.

TypeScript 또한 자바스크립트 대체 언어의 하나로써 **자바스크립트(ES5)의 Superset(상위확장)**이다. C#의 창시자인 덴마크 출신 소프트웨어 엔지니어 Anders Hejlsberg(아네르스 하일스베르)가 개발을 주도한 TypeScript는 Microsoft에서 2012년 발표한 오픈소스로, 정적 타이핑을 지원하며 ES6(ECMAScript 2015)의 클래스, 모듈 등과 ES7의 Decorator 등을 지원한다.



TypeScript는 ES5의 Superset이므로 기존의 자바스크립트(ES5) 문법을 그대로 사용할 수 있다. 또한, ES6의 새로운 기능들을 사용하기 위해 Babel과 같은 별도 트랜스파일러(Transpiler)를 사용하지 않아도 ES6의 새로운 기능을 기존의 자바스크립트 엔진(현재의 브라우저 또는 Node.js)에서 실행할 수 있다.

이후 ECMAScript의 업그레이드에 따른 새로운 기능을 지속적으로 추가할 예정이여서 매년 업그레이드될 ECMAScript의 표준을 따라갈 수 있는 좋은 수단이 될 것이다.



TypeScript의 장점

정적 타입

TypeScript를 사용하는 가장 큰 이유 중 하나는 정적 타입을 지원한다는 것이다. 아래 함수를 살펴보자.

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

위 함수를 정의한 개발자의 의도는 아마도 2개의 숫자 타입 인수를 전달받아 그 합계를 반환하려는 것으로 추측된다. 하지만 코드상으로는 어떤 타입의 인수를 전달하여야 하는지, 어떤 타입의 반환값을 리턴해야 하는지 명확하지 않다. 따라서 위 함수는 아래와 같이 호출될 수 있다.

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

sum('x', 'y'); // 'xy'
```

위 코드는 자바스크립트 문법상 어떠한 문제도 없으므로 자바스크립트 엔진은 아무런 이의 제기없이 위 코드를 실행할 것이다. 이러한 상황이 발생한 이유는 변수나 반환값의 타입을 사전에 지정하지 않는 자바스크립트의 동적 타이핑(Dynamic Typing)에 의한 것이다.

위 함수를 TypeScript의 정적 타입을 사용하여 다시 작성 해보자.

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

sum('x', 'y');
// error TS2345: Argument of type '"x"' is not assignable t
o parameter of type 'number'.
```

TypeScript는 정적 타입을 지원하므로 컴파일 단계에서 오류를 포착할 수 있는 장점이 있다. 명시적인 정적 타입 지정은 개발자의 의도를 명확하게 코드로 기술할 수 있다. 이는 코드의 가독성을 높이고 예측할 수 있게 하며 디버깅을 쉽게 한다.

도구의 지원

TypeScript를 사용하는 이유는 여러가지 있지만 가장 큰 장점은 IDE(통합개발환경)를 포함한 다양한 도구의 지원을 받을 수 있다는 것이다. IDE와 같은 도구에 타입 정보를 제공함으로써 높은 수준의 인텔리센스(IntelliSense), 코드 어시스트, 타입 체크, 리팩토링 등을 지원받을 수 있으며 이러한 도구의 지원은 대규모 프로젝트를 위한 필수 요소이기도 하다.

강력한 객체지향 프로그래밍 지원

인터페이스, 제네릭 등과 같은 강력한 객체지향 프로그래밍 지원은 크고 복잡한 프로젝트의 코드 기반을 쉽게 구성할 수 있도록 도우며, Java, C# 등의 클래스 기반 객체지향 언어에 익숙한 개발자가 자바스크립트 프로젝트를 수행하는 데 진입 장벽을 낮추는 효과도 있다.

ES6 / ES Next 지원

브라우저만 있으면 컴파일러 등의 개발환경 구축없이 바로 사용할 수 있는 ES5와 비교할 때, 개발환경 구축 관점에서 다소 복잡해진 측면이 있지만 현재 ES6를 완전히 지원하지 않고 있는 브라우저를 고려하여 Babel 등의 트랜스파일러를 사용해야 하는 현 상황에서 TypeScript 개발환경 구축에 드는 수고는 그다지 아깝지 않을 것이다. 또한, TypeScript는 아직 ECMAScript 표준에 포함되지는 않았지만 표준화가 유력한 스펙을 선제적으로 도입하므로 새로운 스펙의 유용한 기능을 안전하게 도입하기에 유리하다.

개발환경 구축

TypeScript 파일(.ts)은 브라우저에서 동작하지 않으므로 TypeScript 컴파일러를 이용해 자바스크립트 파일로 변환해야 한다. 이를 컴파일 또는 트랜스파일링이라 한다.

TypeScript 컴파일러를 설치하여 TypeScript 개발환경을 구축하고 TypeScript 컴파일러의 사용 방법에 대해 살펴보도록 하자.

TypeScript 컴파일러 설치 및 사용법

Node.js를 설치하면 npm도 같이 설치된다. 다음과 같이 터미널(윈도우의 경우 커맨드창)에서 npm을 사용하여 TypeScript를 전역에 설치한다.

```
$ npm install -g typescript
```

설치가 완료되었으면 버전을 출력하여 TypeScript의 설치를 확인한다.

```
$ tsc -v
Version 5.4.3
```

TypeScript 컴파일러(tsc)는 TypeScript 파일(.ts)을 자바스크립트 파일로 트랜스파일링 한다.

컴파일은 일반적으로 소스 코드를 바이트 코드로 변환하는 작업을 의미한다. TypeScript 컴파일러는 TypeScript 파일을 자바스크립트 파일로 변환하므로 컴파일보다는 트랜스파일링(Transpiling)이 보다 적절한 표현이다.

트랜스파일링을 실행해보기 위해 아래와 같은 파일을 작성해 보자. 참고로 TypeScript 파일의 확장자는 .ts이다.

```
// person.ts
class Person {
  private name: string;

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

  sayHello() {
    return "Hello, " + this.name;
  }
}

const person = new Person('Lee');

console.log(person.sayHello());
```

트랜스파일링을 실행해 보자. tsc 명령어 뒤에 트랜스파일링 대상 파일명을 지정한다. 이때 확장자 .ts는 생략할 수 있다.

```
$ tsc person
```

트랜스파일링 실행 결과, 같은 디렉터리에 자바스크립트 파일(person.js)이 생성된다.

```
// person.js
var Person = /** @class */ (function () {
    function Person(name) {
        this.name = name;
    }
    Person.prototype.sayHello = function () {
        return "Hello, " + this.name;
    };
    return Person;
}());
var person = new Person('Lee');
console.log(person.sayHello());
```

이때 트랜스파일링된 person.js의 자바스크립트 버전은 ES3이다. 이는 TypeScript 컴파일 타겟 자바스크립트 기본 버전이 ES3이기 때문이다.

만약, 자바스크립트 버전을 변경하려면 컴파일 옵션에 `--target` 또는 `-t`를 사용한다. 현재 tsc가 지원하는 자바스크립트 버전은 'ES3'(default), 'ES5', 'ES2015', 'ES2016', 'ES2017', 'ES2018', 'ES2019', 'ESNEXT'이다. 예를 들어, ES6 버전으로 트랜스파일링을 실행하려면 아래와 같이 옵션을 추가한다.

```
$ tsc person -t ES2015
```

```
// person.js
class Person {
    constructor(name) {
        this.name = name;
    }
    sayHello() {
        return "Hello, " + this.name;
    }
}
```

```
const person = new Person('Lee');
console.log(person.sayHello());
```

트랜스파일링이 성공하여 자바스크립트 파일이 생성되었으면, Node.js REPL을 이용해 트랜스파일링된 person.js를 실행해보자.

```
$ node person
Hello, Lee
```

매번 옵션을 지정하는 것은 번거로우므로 tsc 옵션 설정 파일을 생성하도록 하자.

```
$ tsc --init
message TS6071: Successfully created a tsconfig.json file.
```

아래와 같이 tsc 옵션 설정 파일인 tsconfig.json이 생성된다.

```
{
  "compilerOptions": {
    /* Basic Options */
    // "incremental": true,                   /* Enable inc
remental compilation */
    "target": "es5",                          /* Specify EC
MAScript target version: 'ES3' (default), 'ES5', 'ES2015',
'ES2016', 'ES2017', 'ES2018', 'ES2019' or 'ESNEXT'. */
    "module": "commonjs",                     /* Specify mo
dule code generation: 'none', 'commonjs', 'amd', 'system',
'umd', 'es2015', or 'ESNext'. */
    // "lib": [],
```

tsc 명령어 뒤에 파일명을 지정하면 tsconfig.json이 무시되므로 주의하기 바란다.

```
$ tsc person # ✘ tsconfig.json이 무시된다.
```

tsconfig.json을 적용하려면 아래와 같이 트랜스파일링하도록 한다.

```
$ tsc
```

위와 같이 파일명을 지정하지 않으면 프로젝트 폴더 내의 모든 TypeScript 파일이 모두 트랜스파일링된다. 상속 관계에 있는 두 개의 TypeScript class를 작성해보자.

```
// person.ts
export class Person {
  protected name: string;

  constructor(name: string) {
    this.name = name;
  }
  sayHello() {
    return "Hello, " + this.name;
  }
}
```

```
// student.ts
import { Person } from './person';

class Student extends Person {
  study(): string {
    return `${this.name} is studying.`;
  }
}

const student = new Student('Lee');

console.log(student.sayHello());
console.log(student.study());
```

코드 작성을 마쳤으면 다음 명령으로 두 개의 TypeScript 파일을 한번에 트랜스파일링한다.

```
$ tsc
$ node student
Hello, Lee
Lee is studying.
```

- `-watch` 또는 `w` 옵션을 사용하면 트랜스파일링 대상 파일의 내용이 변경되었을 때 이를 감지하여 자동으로 트랜스파일링이 실행된다.

```
$ tsc --watch
21:23:30 - Compilation complete. Watching for file changes.
```

또는 아래와 같이 tsconfig.json에 watch 옵션을 추가할 수도 있다.

```
{
  // ...
  "watch": true
}
```

student.ts를 변경해 watch 기능이 동작하는지 확인해 보자.

```
// student.ts
import { Person } from './person';

class Student extends Person {
  study(): string {
    return `${this.name} is studying!!`; // studying. → stu
dying!!
  }
}

const student = new Student('Lee');

console.log(student.sayHello());
console.log(student.study());
```

아래와 같이 파일 변경이 감지되고 자동으로 트랜스파일링이 실행된다.

```
[오전 12:40:06] File change detected. Starting incremental c
ompilation...

[오전 12:40:07] Found 0 errors. Watching for file changes.
```

```
$ node student
Hello, Lee
Lee is studying!!
```

컴파일러의 옵션에 대해서는 TypeScript Compiler Options을 참조하기 바란다.