12.2 TypeScript - Visual Studio Code Setup

Visual Studio Code에서의 TypeScript 개발 환경 구축

Visual Studio Code(VSCode)는 마이크로소프트가 제공하는 오픈소스 코드 에디터이다. 마이크로소프트는 TypeScript를 개발한 회사이기도 하여서 VSCode는 TypeScript 지원이 탁월하다. IntelliSense, debugging, Git 등의 기능을 지원하며 다양한 Extension(확장 플러그인)을 제공하여 자신의 프로젝트에 맞는 개발 환경을 쉽게 구축할 수 있다.

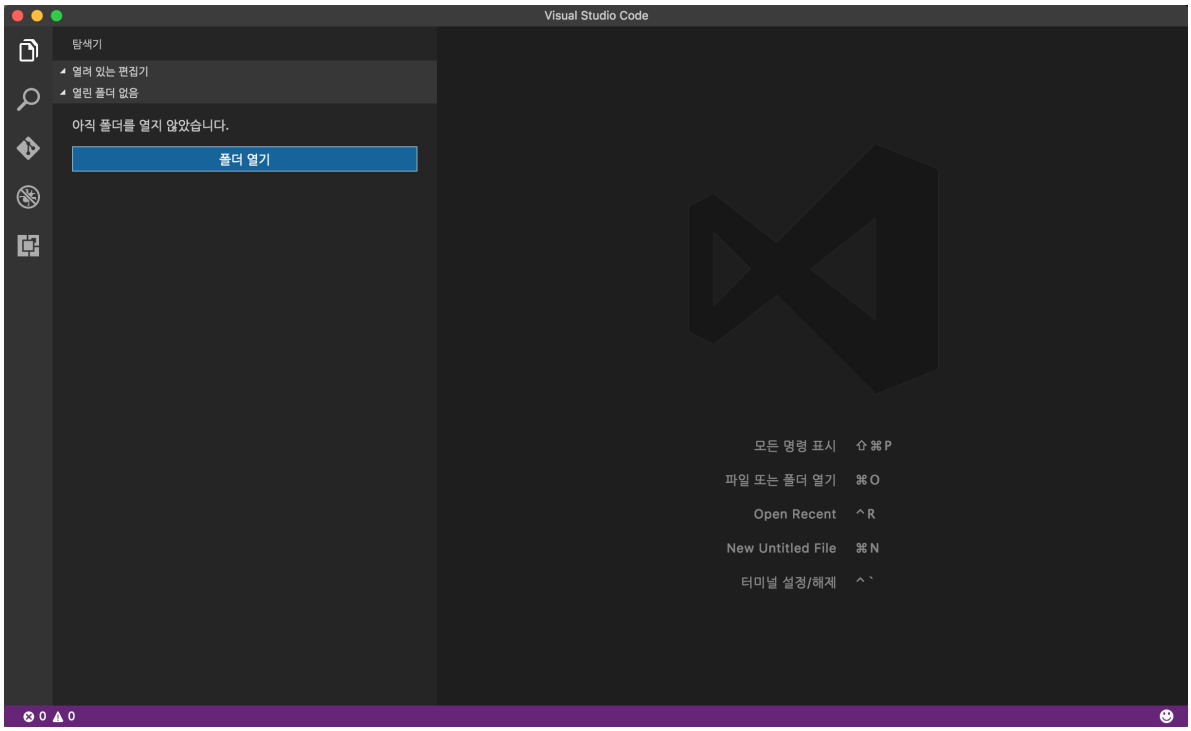
#1. Visual Studio Code 설치

VSCode 설치는 간단하다. 자신의 OS에 맞는 인스톨러를 다운로드하여 설치하도록 하자.

- Running VS Code on Windows

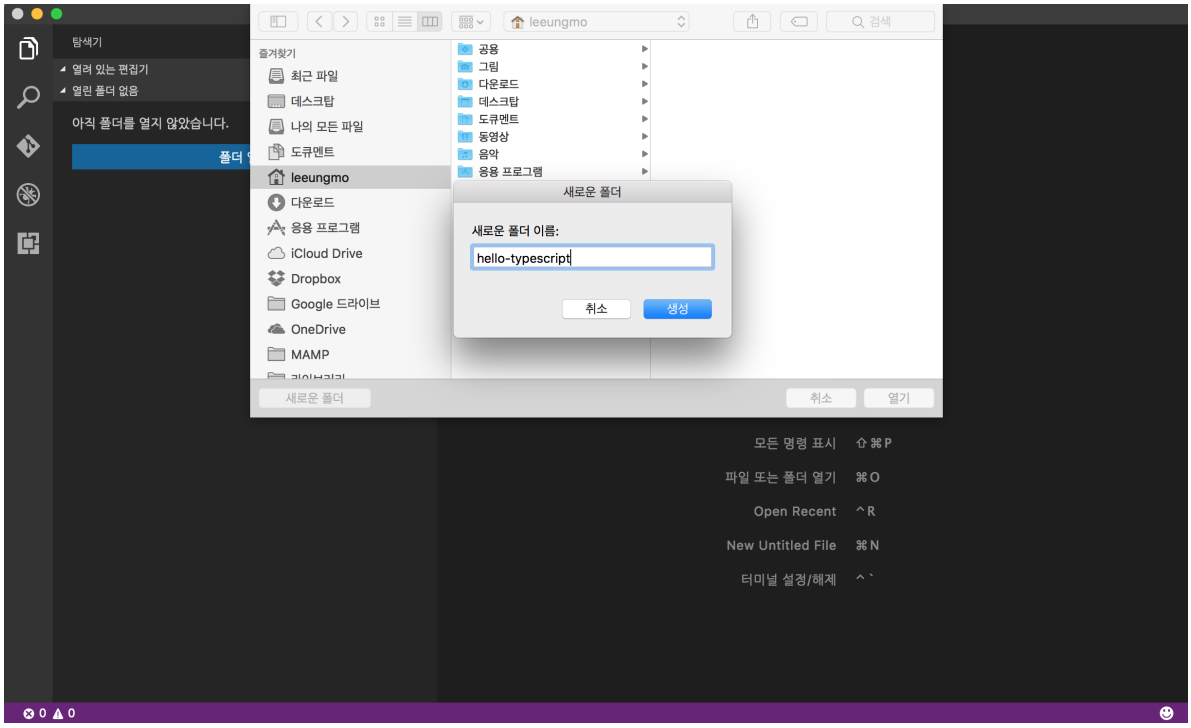
- Running VS Code on Mac

VSCode를 설치가 완료되었으면 적당한 위치에 프로젝트 폴더를 생성한다. 좌측 맨위의 파일 모양의 “탐색기” 아이콘을 선택하면 프로젝트 폴더를 선택할 수 있는 버튼이 표시된다.



탐색기 선택

“폴더 열기” 버튼을 클릭하고 적당한 위치에 프로젝트 폴더를 생성한다.



프로젝트 폴더 성성

2. tsconfig.json

컴파일할 때마다 다양한 옵션을 반복적으로 지정하는 것은 번거러운 일이므로 tsconfig.json을 사용하는 편이 좋다. tsconfig.json은 TypeScript를 위한 프로젝트 단위의 환경 파일로써 컴파일 옵션과 컴파일 대상에 대한 설정 등을 기술한 것이다.

`compilerOptions` 프로퍼티에는 컴파일 옵션을 설정한다. 생략한 경우에는 기본 컴파일 옵션이 사용된다.

JSON

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "sourceMap": true
  }
}
```

컴파일 대상 파일을 지정하기 위해서 `files` 또는 `include` 프로퍼티를 사용한다. 만약 files 프로퍼티를 정의하였다면 include 프로퍼티는 무시된다.

`files` 프로퍼티에는 컴파일 대상 파일의 상대 경로 또는 절대 경로를 명시적으로 설정한다.

JSON

```
{
  "files": [
    "src/file1.ts",
    "src/file2.ts"
  ]
}
```

`include` 프로퍼티에는 컴파일 대상 파일 리스트를 설정한다. `exclude` 프로퍼티에는 컴파일 대상에서 제외할 파일 리스트를 설정한다.

JSON

```
{
  "include": [
    "src/**/*"
  ],
  "exclude": [
    "node_modules",
    "**/*.spec.ts"
  ]
}
```

`src/\*\*/\*` 는 src 폴더 내에 있는 모든 서브 폴더 내의 모든 파일(.ts, .tsx)을 의미한다. 컴파일 옵션 `"allowJs": true` 를 설정하면 .js와 .jsx 파일도 컴파일 대상이 된다.

이제 프로젝트 폴더에 tsconfig.json 파일을 생성해 보자. tsconfig.json에 아래와 같이 컴파일 설정을 편집한다.

JSON

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "sourceMap": true
  }
}
```

이제 간단한 TypeScript 코드를 작성해보자. 파일명은 HelloWorld.ts로 지정한다.

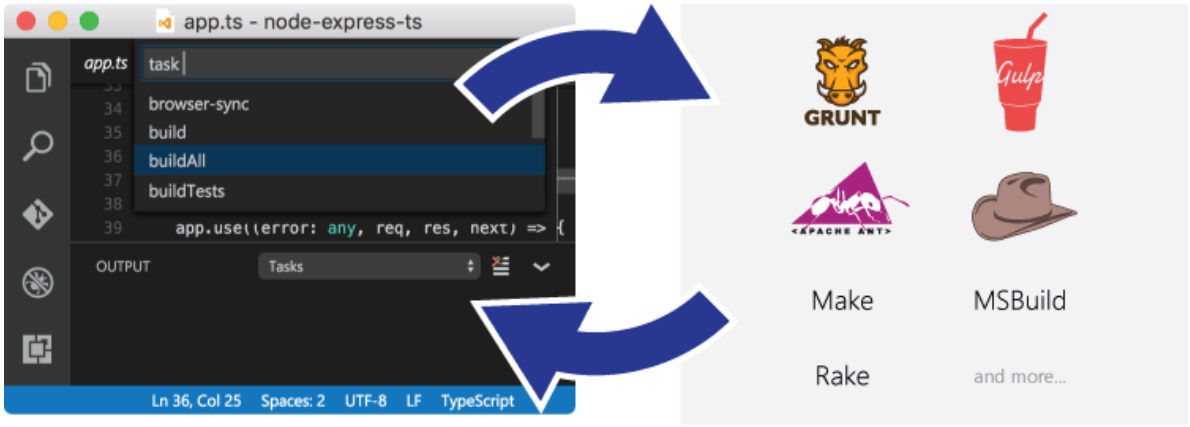
TYPESCRIPT

```
class Startup {
  public static main(): number {
    console.log('Hello World');
    return 0;
  }
}

Startup.main();
```

3. Task runner

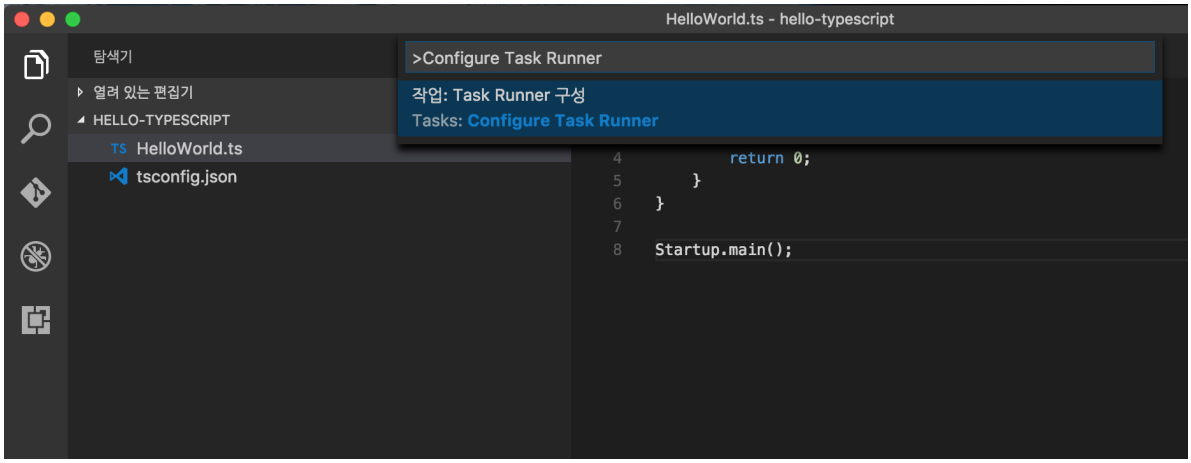
위 TypeScript 코드를 자바스크립트로 트랜스파일링하기 위해서는 TypeScript 컴파일러를 실행시켜야한다. 컴파일러는 VSCode 외부에 존재하므로 VSCode와 TypeScript 컴파일러 간의 연동이 필요하다.

VSCode는 Task runner로 외부의 툴을 VSCode와 연동시킬 수 있다. CLI로 실행되는 툴들을 VSCode에서 실행시킬 수 있는 수 있도록 하는 것이다.



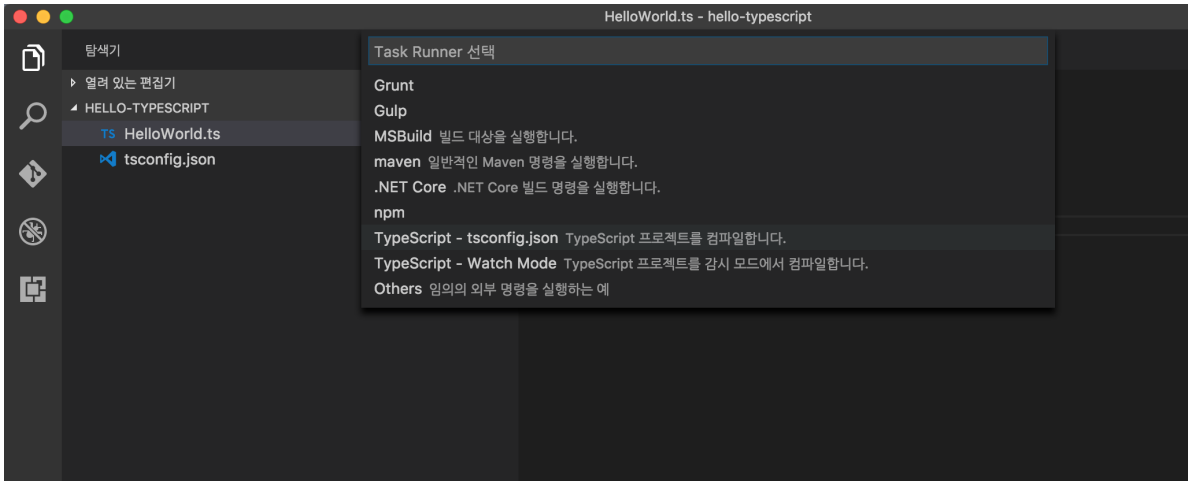
task runner

`Ctrl + Shft + P(⇧⌘P)` 단축기 또는 메뉴의 보기 > 명령 팔레트를 선택하고 “Configure Task Runner”를 입력한다.



명령 팔레트에서 “Configure Task Runner” 입력

“TypeScript - tsconfig.json”을 선택한다.



"TypeScript - tsconfig.json" 선택

.vscode라는 숨겨진 폴더에 아래와 같은 `tasks.json` 파일이 생성된다.

JSON

```
{
  "version": "0.1.0",
  "command": "tsc",
  "isShellCommand": true,
  "args": ["-p", "."],
  "showOutput": "silent",
  "problemMatcher": "$tsc"
}
```

이제 ts 파일을 js 파일로 컴파일 해보자. `Ctrl + Shft + B(⇧⌘B)` 단축키를 누르면 HelloWorld.js와 HelloWorld.js.map이 생성된다.

터미널에서 트랜스파일링된 HelloWorld.js를 실행해보자.

BASH

```
$ node HelloWorld.js
Hello World
```

보기 > 통합 터미널 (^`)을 선택하면 VSCode의 내장 터미널을 사용할 수도 있다.

개발시에는 코드가 빈번히 변경되므로 코드의 변경을 감시하도록 task runner의 설정을 변경해 보자.

아래와 같이 `tasks.json` 파일을 수정한다.

JSON

```
{
  "version": "0.1.0",
  "command": "tsc",
  "isShellCommand": true,
  "args": ["-w", "-p", "."],
  "showOutput": "always",
  "isWatching": true,
  "problemMatcher": "$tsc-watch"
}
```

`Ctrl + Shft + B(⇧⌘B)` 단축키로 다시 빌드를 수행한다.

BASH

```
9:09:09 PM - Compilation complete. Watching for file changes.
```

이제 파일의 변경을 감시하기 시작하며 변경이 발생하면 자동으로 재빌드를 수행한다. ts 파일을 수정해 보자.

TYPESCRIPT

```
class Startup {
  public static main(): number {
    console.log('Hello Angular2');
    return 0;
  }
}

Startup.main();
```

BASH

```
9:17:01 PM - File change detected. Starting incremental compilation...
9:17:02 PM - Compilation complete. Watching for file changes.
```

터미널에서 트랜스파일링된 HelloWorld.js를 실행하여 파일 변경이 반영된 것을 확인한다.

BASH

```
$ node HelloWorld.js
Hello Angular2
```

4. 외부 라이브러리의 사용을 위한 TypeScript Definition 설치

TypeScript를 사용하는 이유는 여러가지 있지만 가장 큰 장점은 다양한 도구의 지원을 받을 수 있다는 것이다. TypeScript는 정적 타입을 지원하므로 높은 수준의 IntelliSense나 리팩토링 등을 지원하며 이러란 도구의 지원은 대규모 프로젝트를 위한 필수적 요소이기도 하다.

프로젝트 내에는 필수적으로 다양한 라이브러리가 포함되는데 이 라이브러리들은 JavaScript로 작성되어있다. TypeScript는 ES5의 Superset(상위확장)이므로 JavaScript를 그대로 사용할 수 있다. 하지만 정적 타입이 없는 JavaScript를 그대로 사용하면 VSCode에서 제공하는 IntelliSense와 같은 다양한 도구의 지원을 받을 수 없다.

따라서 외부 JavaScript 라이브러리에 대해서도 타입체크를 수행하려면 해당 라이브러리의 타입이 정의되어 있는 **정의 파일**(Definition file)을 제공해야 한다.

라이브러리의 정의 파일을 직접 수작업으로 만드는 것은 어려운 일이다. 다행스럽게도 npm에서 정의 파일을 설치할 수 있다.

위의 예제에서 유틸리티 라이브러리 lodash를 사용해 보자.

우선 lodash를 설치한다.

BASH

```
$ npm init -y
$ npm install lodash --save
```

npm에서 lodash 정의 파일을 설치한다.

BASH

```
$ npm install @types/lodash --save-dev
```

ts 파일을 수정해 보자.

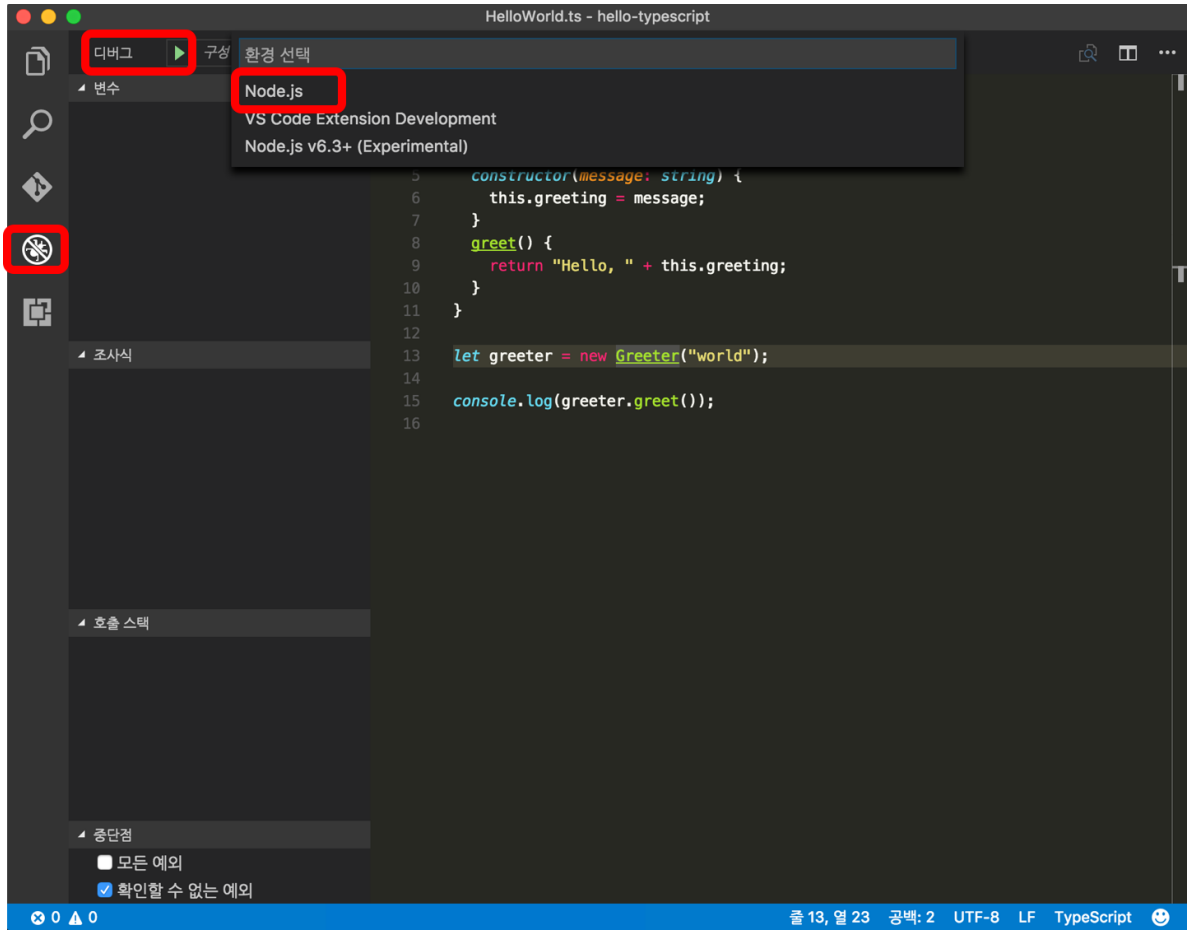
TYPESCRIPT

```typescript
import * as _ from "lodash";

class Startup {
  public static main(): number {
    const group = _.groupBy(['one', 'two', 'three'], 'length');
    console.log(group); // ⇒ { '3': ['one', 'two'], '5': ['three'] }
    return 0;
  }
}

Startup.main();
```

이제 lodash 라이브러리에 대해 IntelliSense가 작동하는 것을 확인할 수 있다.

#5. 디버깅

디버깅을 위해서는 좌측 상단의 벌레 모양 아이콘을 클릭한 후 화면 상단의 디버그 버튼을 클릭하면 `launch.json` 파일이 생성된다.



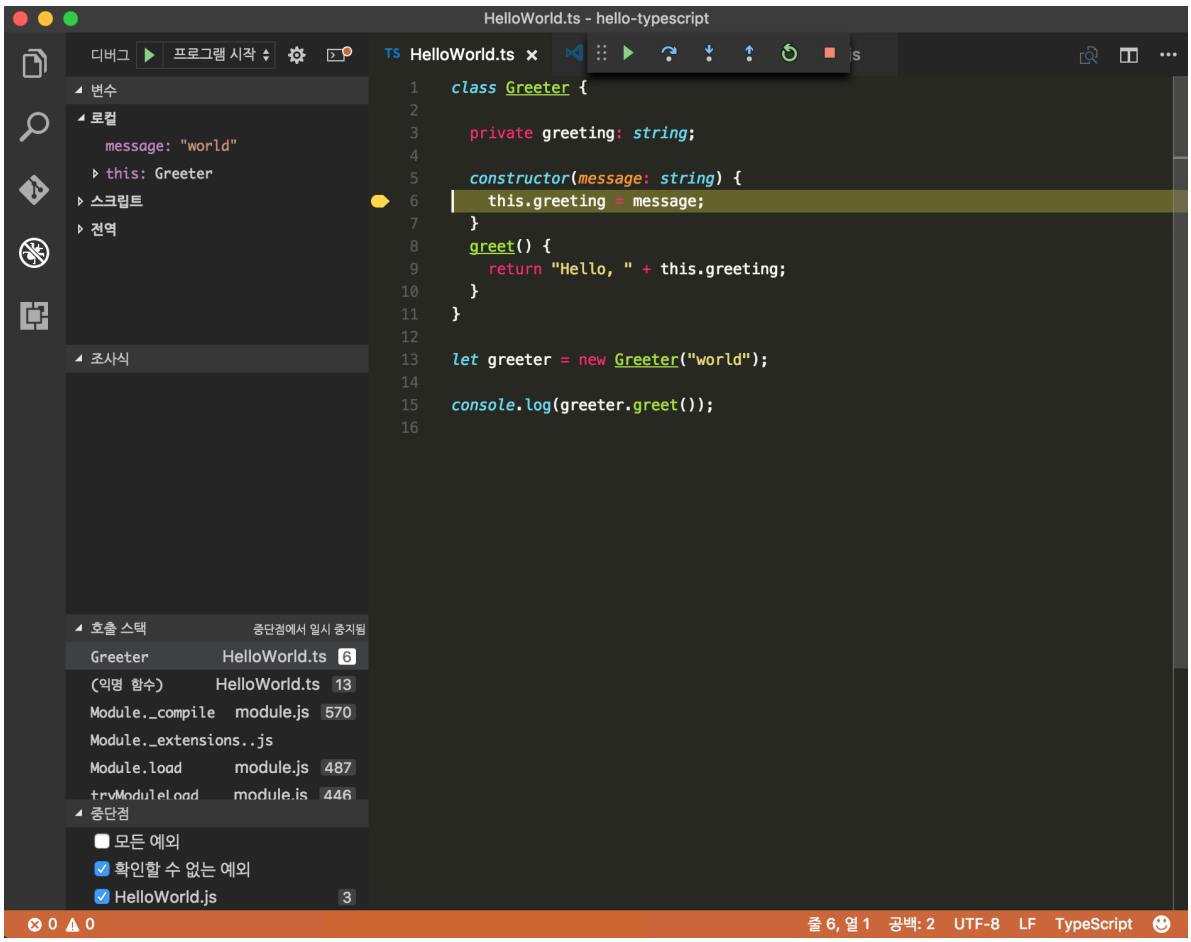
configurations 프로퍼티의 program 프로퍼티값을 디버깅할 파일명으로 변경한다.

JSON

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "프로그램 시작",
      "program": "${workspaceRoot}/HelloWorld.ts",
      "cwd": "${workspaceRoot}",
      "outFiles": [],
      "sourceMaps": true
    },
    {
      "type": "node",
      "request": "attach",
      "name": "프로세스에 연결",
      "port": 5858,
      "outFiles": [],
      "sourceMaps": true
```

```
      }
    ]
}
```

중단점을 설정하고 디버그 버튼을 클릭하면 디버깅이 시작된다.



VSCode에서의 TypeScript의 사용에 대한 보다 자세한 내용은 Visual Studio Code: Editing TypeScript을 참조하기 바란다.

Reference

- Visual Studio Code(VSCode)
- Visual Studio Code: tsconfig.json
- Visual Studio Code: Task runner
- Visual Studio Code: Editing TypeScript
- Typings