10장 네임스페이와 모듈

TypeScript에서 모듈과 네임스페이스를 사용하여 코드를 구성하는 다양한 방법이 있다. 또한 네임스페이스와 모듈에 관한 몇 가지 고급 주제와 TypeScript에서 네임스페이스와 모듈을 사용할 때 흔히 마주치는 위험성을 살펴 보아야 한다.

모듈 사용하기 (Using Modules)

모듈에는 코드와 선언 둘 다 포함될 수 있습니다.

모듈은 또한 모듈 로더(예: CommonJs/Require.js)에 대한 의존성이나 ES 모듈이 지원하는 런타임을 가지고 있습니다. 모듈은 더 나은 코드 재사용성을 위해, 강력한 고립성과 번들링을 위한 향상된 도구를 제공합니다.

또한 Node.js 애플리케이션의 경우 모듈이 기본적인 방법이며, 코드를 구조화하는 데 있어 권장하는 접근법이라는 점을 유의해야 합니다.

ECMAScript 2015부터, 모듈은 언어에서 기본적으로 내재한 부분이며, 모든 호환 엔진 구현은 모듈을 지원해야 합니다. 따라서, 새로운 프로젝트의 경우 코드를 구성하는 방법으로 모듈을 권장합니다.

네임스페이스 사용하기 (Using Namespaces)

네임스페이스는 코드를 구성하는 TypeScript만의 고유한 방법입니다. 네임스페이스는 간단히 전역 네임스페이스에서 JavaScript 객체로 이름 붙여집니다. 이러한 점 덕분에 네임스페이스를 아주 단순한 구조로 사용할 수 있습니다. 모듈과 달리, 여러 개의 파일을 포괄할 수 있으며, `--outFile`을 사용해 연결할 수 있습니다. 네임스페이스는 웹 애플리케이션에서 코드를 구조화하기에 좋은 방법이며, 모든 의존성은 HTML 페이지의 `<script>` 태그로 포함합니다.

특히 대규모 애플리케이션의 경우, 이 방법은 모든 전역 네임스페이스가 오염되는 경우와 마찬가지로 컴포넌트의 의존성을 식별하기 힘들게 만들 수 있습니다.

네임스페이스와 모듈의 위험성 (Pitfalls of Namespaces and Modules)

여기서는 네임스페이스와 모듈을 사용할 때 자주 발생하는 다양한 위험성과 그 해결책을 알아보겠습니다.

`/// <reference>`를 사용한 모듈

일반적인 실수는 모듈 파일을 참조하기 위해 `import`문 대신 `/// <reference ... />` 구문을 사용하는 것입니다. 이 둘의 차이를 이해하기 위해, 우선 `import` 경로에 위치한 모듈에 대한 타입 정보를 컴파일러가 어떻게 찾아내는지를 이해해야 합니다. (예를 들어, `import x from "...";`, `import x = require("...");` 등 안의 `...`)

컴파일러는 `.ts`, `.tsx`를 찾은 다음 적절한 경로에 위치한 `.d.ts`를 찾습니다. 만약 특정 파일을 찾지 못한다면, 컴파일러는 *앰비언트 모듈(ambient module) 선언*을 탐색할 것입니다. `.d.ts` 파일안에 이것들을 선언해야 한다는 점을 기억하세요.

- `myModules.d.ts`

```ts
// 모듈이 아닌 .d.ts 파일 또는 .ts 파일:
declare module "SomeModule" {
    export function fn(): string;
}
```

- `myOtherModule.ts`

```ts
/// <reference path="myModules.d.ts" />
import * as m from "SomeModule";
```

위의 reference 태그는 앰비언트 모듈(ambient module) 선언이 포함된 선언 파일의 위치를 지정하는 데 필요합니다. 이 방법은 여러 TypeScript 샘플에서 사용하는 `node.d.ts` 파일을 사용하는 방법입니다.

불필요한 네임스페이스 (Needless Namespacing)

네임스페이스를 사용하던 프로그램을 모듈로 변경하면, 파일은 다음과 같은 모습이 되기 쉽습니다:

- `shapes.ts`

```ts
export namespace Shapes {
    export class Triangle { /* ... */ }
    export class Square { /* ... */ }
}
```

최상위 모듈 `Shapes`는 아무런 의미 없이 `Triangle`과 `Square`을 감싸고 있습니다. 이런 점은 모듈 사용자에게 혼동과 짜증을 유발합니다:

- `shapeConsumer.ts`

```ts
import * as shapes from "./shapes";
let t = new shapes.Shapes.Triangle(); // shapes.Shapes?
```

TypeScript 모듈의 중요한 특징 중 하나는 서로 다른 두 개의 모듈이 절대 같은 스코프 안에 이름을 제공하지 않는다는 점입니다. 모듈 사용자가 어떤 이름을 할당할지를 결정하기 때문에, 네임스페이스 내부에서 내보내는 심벌을 미리 감싸줄 필요가 없습니다.

모듈 내용의 네임스페이스를 설정하지 않아도 되는 이유를 다시 말하면, 네임스페이스를 지정하는 일반적인 목적은 구조의 논리적 그룹을 제공하고 이름 충돌을 방지하기 위함입니다. 모듈 파일이 이미 스스로 논리적 그룹을 형성하고 있기 때문에, 최상위 이름은 이를 가져오는 코드에 의해 정의되고, 내보내는 객체를 위한 추가적인 모듈 계층을 사용할 필요가 없습니다.

다음은 수정된 예입니다:

- `shapes.ts`

```ts
export class Triangle { /* ... */ }
export class Square { /* ... */ }
```

- shapeConsumer.ts

```
import * as shapes from "./shapes";
let t = new shapes.Triangle();
```

모듈의 트레이드-오프 (Trade-offs of Modules)

JS 파일과 모듈이 일대일 대응인 것처럼, TypeScript는 모듈 소스 파일과 이 파일에서 생성된 JS 파일도 일대일 대응입니다. 이러한 특성 때문에 어떤 모듈 시스템을 사용하느냐에 따라서 여러 모듈 소스 파일을 합치는 작업이 불가능할 수 있습니다. 예를 들어, `commonjs` 또는 `umd`를 대상으로 하는 동안에는 `outFile` 옵션을 사용할 수 없지만, TypeScript 1.8 이후부터, `amd` 또는 `system`를 대상으로 하는 경우에는 `outFile` 옵션을 사용할 수 있게 되었습니다.