ECMAScript 2015부터 JavaScript에는 모듈 개념이 있습니다. TypeScript는 이 개념을 공유합니다.

모듈은 전역 스코프가 아닌 자체 스코프 내에서 실행됩니다; 즉 모듈 내에서 선언된 변수, 함수, 클래스 등은 `export` 양식 중 하나를 사용하여 명시적으로 export 하지 않는 한 모듈 외부에서 보이지 않습니다. 반대로 다른 모듈에서 export 한 변수, 함수, 클래스, 인터페이스 등을 사용하기 위해서는 `import` 양식 중 하나를 사용하여 import 해야 합니다.

모듈은 선언형입니다; 모듈 간의 관계는 파일 수준의 imports 및 exports 관점에서 지정됩니다.

모듈은 모듈 로더를 사용하여 다른 모듈을 import 합니다. 런타임 시 모듈 로더는 모듈을 실행하기 전에 모듈의 모든 의존성을 찾고 실행해야 합니다. JavaScript에서 사용하는 유명한 모듈 로더로는 CommonJS 모듈 용 Node.js의 로더와 웹 애플리케이션의 AMD 모듈 용 RequireJS 로더가 있습니다.

ECMAScript 2015와 마찬가지로 TypeScript는 최상위 수준의 `import` 혹은 `export`가 포함된 모든 파일을 모듈로 간주합니다. 반대로 최상위 수준의 `import` 혹은 `export` 선언이 없는 파일은 전역 스코프에서 사용할 수 있는 스크립트로 처리됩니다 (모듈에서도 마찬가지).

Export

선언 export 하기 (Exporting a declaration)

`export` 키워드를 추가하여 모든 선언 (변수, 함수, 클래스, 타입 별칭, 인터페이스)를 export 할 수 있습니다.

StringValidator.ts

```ts
export interface StringValidator {
    isAcceptable(s: string): boolean;
}
```

ZipCodeValidator.ts

```ts
import { StringValidator } from "./StringValidator";

export const numberRegexp = /^[0-9]+$/;

export class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}
```

Export 문 (Export statements)

Export 문은 사용자를 위해 export 할 이름을 바꿔야 할 때 편리합니다. 위의 예제는 다음과 같이 작성할 수 있습니다:

```ts
class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}
export { ZipCodeValidator };
export { ZipCodeValidator as mainValidator };
```

Re-export 하기 (Re-exports)

종종 모듈은 다른 모듈을 확장하고 일부 기능을 부분적으로 노출합니다. Re-export 하기는 지역적으로 import 하거나, 지역 변수를 도입하지 않습니다.

ParseIntBasedZipCodeValidator.ts

```
export class ParseIntBasedZipCodeValidator {
    isAcceptable(s: string) {
        return s.length === 5 && parseInt(s).toString() === s;
    }
}

// 기존 validator의 이름을 변경 후 export
export {ZipCodeValidator as RegExpBasedZipCodeValidator} from "./ZipCodeValidator";
```

선택적으로, 하나의 모듈은 하나 혹은 여러 개의 모듈을 감쌀 수 있고, `export * from "module"` 구문을 사용해 export 하는 것을 모두 결합할 수 있습니다.

AllValidators.ts

```
export * from "./StringValidator"; // 'StringValidator' 인터페이스를 내보냄
export * from "./ZipCodeValidator";  // 'ZipCodeValidator' 와 const 'numberRegexp' 클래스를 내보냄
export * from "./ParseIntBasedZipCodeValidator"; // 'ParseIntBasedZipCodeValidator' 클래스를 내보냄
                                                 // 'ZipCodeValidator.ts' 모듈 에 있는
                                                 // 'ZipCodeValidator' 클래스를
                                                 // 'RegExpBasedZipCodeValidator' 라는 별칭으로 다시 내보냄
```

Import

import는 모듈에서 export 만큼 쉽습니다. export 한 선언은 아래의 `import` 양식 중 하나를 사용하여 import 합니다:

모듈에서 단일 export를 import 하기 (Import a single export from a module)

```
import { ZipCodeValidator } from "./ZipCodeValidator";

let myValidator = new ZipCodeValidator();
```

이름을 수정해서 import 할 수 있습니다.

```
import { ZipCodeValidator as ZCV } from "./ZipCodeValidator";
let myValidator = new ZCV();
```

전체 모듈을 단일 변수로 import 해서, 모듈 exports 접근에 사용하기 (Import the entire module into a single variable, and use it to access the module exports)

```
import * as validator from "./ZipCodeValidator";
let myValidator = new validator.ZipCodeValidator();
```

부수효과만을 위해 모듈 import 하기 (Import a module for side-effects only)

권장되지는 않지만, 일부 모듈은 다른 모듈에서 사용할 수 있도록 일부 전역 상태로 설정합니다. 이러한 모듈은 어떤 exports도 없거나, 사용자가 exports에 관심이 없습니다. 이러한 모듈을 import 하기 위해, 다음처럼 사용하세요:

```
import "./my-module.js"
```

타입 import 하기 (Importing Types)

TypeScript 3.8 이전에는 `import`를 사용하여 타입을 import 할 수 있었습니다. TypeScript 3.8에서는 `import` 문 혹은 `import type`을 사용하여 타입을 import 할 수 있습니다.

```
// 동일한 import를 재사용하기
import {APIResponseType} from "./api";

// 명시적으로 import type을 사용하기
import type {APIResponseType} from "./api";
```

`import type`은 항상 JavaScript에서 제거되며, 바벨 같은 도구는 `isolatedModules` 컴파일러 플래그를 통해 코드에 대해 더 나은 가정을 할 수 있습니다. 3.8 릴리즈 정보에서 더 많은 정보를 읽을 수 있습니다.

Default exports

각 모듈은 선택적으로 `default` export를 export 할 수 있습니다. default export는 `default` 키워드로 표시됩니다; 모듈당 하나의 `default` export만 가능합니다. `default` export는 다른 import 양식을 사용하여 import 합니다.

`default` exports는 정말 편리합니다. 예를 들어 jQuery와 같은 라이브러리는 `jQuery` 혹은 `$`와 같은 default export를 가질 수 있으며, `$`나 `jQuery`와 같은 이름으로 import할 수 있습니다.

JQuery.d.ts

```
declare let $: JQuery;
export default $;
```

App.ts

```
import $ from "jquery";

$("button.continue").html( "Next Step..." );
```

클래스 및 함수 선언은 default exports로 직접 작성될 수 있습니다. default export 클래스 및 함수 선언 이름은 선택사항 입니다.

ZipCodeValidator.ts

```
export default class ZipCodeValidator {
    static numberRegexp = /^[0-9]+$/;
    isAcceptable(s: string) {
        return s.length === 5 && ZipCodeValidator.numberRegexp.test(s);
    }
}
```

Test.ts

```
import validator from "./ZipCodeValidator";

let myValidator = new validator();
```

혹은

StaticZipCodeValidator.ts

```
const numberRegexp = /^[0-9]+$/;

export default function (s: string) {
    return s.length === 5 && numberRegexp.test(s);
}
```

Test.ts

```
import validate from "./StaticZipCodeValidator";

let strings = ["Hello", "98052", "101"];

// validate 함수 사용하기
strings.forEach(s => {
  console.log(`"${s}" ${validate(s) ? "matches" : "does not match"}`);
});
```

`default` exports는 값도 가능합니다:

OneTwoThree.ts

```
export default "123";
```

Log.ts

```
import num from "./OneTwoThree";

console.log(num); // "123"
```

x로 모두 export 하기 (Export all as x)

TypeScript 3.8에서는 다음 이름이 다른 모듈로 re-export 될 때 단축어처럼 `export * as ns`를 사용할 수 있습니다:

```
export * as utilities from "./utilities";
```

모듈에서 모든 의존성을 가져와 export한 필드로 만들면, 다음과 같이 import할 수 있습니다:

```
import { utilities } from "./index";
```

`export =`와 `import = require()` (`export =` and `import = require()`)

CommonJS와 AMD 둘 다 일반적으로 모듈의 모든 exports를 포함하는 `exports` 객체의 개념을 가지고 있습니다.

또한 `exports` 객체를 사용자 정의 단일 객체로 대체하는 기능도 지원합니다. default exports는 이 동작에서 대체 역할을 합니다; 하지만 둘은 호환되지는 않습니다. TypeScript는 기존의 CommonJS와 AMD 워크플로우를 모델링 하기 위해 `export =`를 지원합니다.

`export =` 구문은 모듈에서 export되는 단일 객체를 지정합니다. 클래스, 인터페이스, 네임스페이스, 함수 혹은 열거형이 될 수 있습니다.

`export =`를 사용하여 모듈을 export할 때, TypeScript에 특정한 `import module = require("module")`를 사용하여 모듈을 가져와야 합니다.

ZipCodeValidator.ts

```
let numberRegexp = /^[0-9]+$/;
class ZipCodeValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}
export = ZipCodeValidator;
```

Test.ts

```
import zip = require("./ZipCodeValidator");

// 시험용 샘플
let strings = ["Hello", "98052", "101"];

// 사용할 Validators
let validator = new zip();

// 각 문자열이 각 validator를 통과했는지 보여줍니다
strings.forEach(s => {
  console.log(`"${ s }" - ${ validator.isAcceptable(s) ? "matches" : "does not match" }`);
});
```

모듈을 위한 코드 생성 (Code Generation for Modules)

컴파일 중에는 지정된 모듈 대상에 따라 컴파일러는 Node.js (CommonJS), require.js (AMD), UMD, SystemJS, 또는 ECMAScript 2015 native modules (ES6) 모듈-로딩 시스템에 적합한 코드를 생성합니다. 생성된 코드의 `define`, `require` 그리고 `register` 호출 기능에 대한 자세한 정보는 각 모듈 로더의 문서를 확인하세요.

이 간단한 예제는 import 및 export 하기 중에 사용된 이름이 모듈 로딩 코드로 변환되는 방법을 보여줍니다.

SimpleModule.ts

```
import m = require("mod");
export let t = m.something + 1;
```

AMD / RequireJS SimpleModule.js

```
define(["require", "exports", "./mod"], function (require, exports, mod_1) {
    exports.t = mod_1.something + 1;
});
```

CommonJS / Node SimpleModule.js

```
var mod_1 = require("./mod");
exports.t = mod_1.something + 1;
```

UMD SimpleModule.js

```
(function (factory) {
    if (typeof module === "object" && typeof module.exports === "object") {
        var v = factory(require, exports); if (v !== undefined) module.exports = v;
    }
    else if (typeof define === "function" && define.amd) {
        define(["require", "exports", "./mod"], factory);
    }
})(function (require, exports) {
    var mod_1 = require("./mod");
    exports.t = mod_1.something + 1;
});
```

System SimpleModule.js

```
System.register(["./mod"], function(exports_1) {
    var mod_1;
    var t;
    return {
        setters:[
            function (mod_1_1) {
                mod_1 = mod_1_1;
            }],
        execute: function() {
            exports_1("t", t = mod_1.something + 1);
        }
    }
});
```

Native ECMAScript 2015 modules SimpleModule.js

```
import { something } from "./mod";
export var t = something + 1;
```

간단한 예제 (Simple Example)

아래에서는 각 모듈에서 단일 이름으로 export 하기 위해 이전 예제에서 사용한 Validator 구현을 통합합니다.

컴파일 하려면, 명령 줄에서 모듈 대상을 지정해야 합니다. Node.js의 경우, `--module commonjs`를 사용하세요; require.js의 경우 `--module amd`를 사용하세요. 예를 들면:

```
tsc --module commonjs Test.ts
```

컴파일이 되면, 각 모듈은 별도의 `.js`파일이 됩니다. 참조 태그와 마찬가지로, 컴파일러는 `import`문을 따라 의존적인 파일들을 컴파일 합니다.

Validation.ts

```
export interface StringValidator {
    isAcceptable(s: string): boolean;
}
```

LettersOnlyValidator.ts

```
import { StringValidator } from "./Validation";

const lettersRegexp = /^[A-Za-z]+$/;

export class LettersOnlyValidator implements StringValidator {
    isAcceptable(s: string) {
        return lettersRegexp.test(s);
    }
}
```

ZipCodeValidator.ts

```
import { StringValidator } from "./Validation";

const numberRegexp = /^[0-9]+$/;

export class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}
```

Test.ts

```
import { StringValidator } from "./Validation";
import { ZipCodeValidator } from "./ZipCodeValidator";
import { LettersOnlyValidator } from "./LettersOnlyValidator";

// 시험용 샘플
let strings = ["Hello", "98052", "101"];

// 사용할 validator
let validators: { [s: string]: StringValidator; } = {};
validators["ZIP code"] = new ZipCodeValidator();
validators["Letters only"] = new LettersOnlyValidator();

// 각 문자열이 validator를 통과하는지 보여줌
strings.forEach(s => {
    for (let name in validators) {
        console.log(`"${ s }" - ${ validators[name].isAcceptable(s) ? "matches" : "does not match" } ${ name }`);
    }
});
```

선택적 모듈 로딩과 기타 고급 로딩 시나리오 (Optional Module Loading and Other Advanced Loading Scenarios)

상황에 따라 특정 조건에서만 모듈을 로드하도록 만들 수 있습니다. TypeScript에서는 아래에 있는 패턴을 사용하여 이 시나리오와 다른 고급 로딩 시나리오를 구현하여 타입의 안전성을 잃지 않고 모듈 로더를 직접 호출할 수 있습니다.

컴파일러는 노출된 JavaScript 안에서 각 모듈의 사용 여부를 감지합니다. 모듈 식별자가 표현식이 아닌 타입 표시로만 사용된다면 그 모듈에 대한 `require` 호출은 발생하지 않습니다. 사용하지 않는 참조를 제거하면 성능을 최적화할 수 있으며, 해당 모듈을 선택적으로 로딩 할 수 있습니다.

이 패턴의 핵심 아이디어는 `import id = require("...")` 문을 통해 모듈로 노출된 타입에 접근이 가능하다는 것입니다. 아래 `if` 블록에 보이는 것처럼, 모듈 로더는 (`require`을 통해) 동적으로 호출됩니다. 이 기능은 참조-제거 최적화를 활용하므로 필요할 때만 모듈을 로드할 수 있습니다. 해당 패턴이 동작하려면 `import`를 통해 정의된 심벌은 오직 타입 위치(즉, JavaScript로 방출되는 위치에서는 사용 안 함)에서만 사용되는 것이 중요합니다.

타입 안전성을 유지하기 위해, `typeof` 키워드를 사용할 수 있습니다. `typeof` 키워드는 타입 위치에서 사용될 때는 값의 타입, 이 경우에는 모듈의 타입을 생성합니다.

Node.js에서 동적 모듈 로딩 (Dynamic Module Loading in Node.js)

```
declare function require(moduleName: string): any;

import { ZipCodeValidator as Zip } from "./ZipCodeValidator";

if (needZipValidation) {
    let ZipCodeValidator: typeof Zip = require("./ZipCodeValidator");
    let validator = new ZipCodeValidator();
    if (validator.isAcceptable("...")) { /* ... */ }
}
```

샘플: require.js에서 동적 모듈 로딩 (Sample: Dynamic Module Loading in require.js)

```
declare function require(moduleNames: string[], onLoad: (...args: any[]) => void): void;

import * as Zip from "./ZipCodeValidator";

if (needZipValidation) {
    require(["./ZipCodeValidator"], (ZipCodeValidator: typeof Zip) => {
        let validator = new ZipCodeValidator.ZipCodeValidator();
        if (validator.isAcceptable("...")) { /* ... */ }
    });
}
```

샘플: System.js에서 동적 모듈 로딩 (Sample: Dynamic Module Loading in System.js)

```
declare const System: any;

import { ZipCodeValidator as Zip } from "./ZipCodeValidator";

if (needZipValidation) {
    System.import("./ZipCodeValidator").then((ZipCodeValidator: typeof Zip) => {
        var x = new ZipCodeValidator();
        if (x.isAcceptable("...")) { /* ... */ }
    });
}
```

다른 JavaScript 라이브러리와 함께 사용하기 (Working with Other JavaScript Libraries)

TypeScript로 작성되지 않은 라이브러리의 형태를 설명하려면, 라이브러리를 노출하는 API를 선언해야 합니다.

우리는 구현을 정의하지 않은 선언을 "ambient"라고 부릅니다. 이 선언들은 일반적으로 `.d.ts` 파일에 정의되어 있습니다. C/C++에 익숙하다면, `.h` 파일이라고 생각할 수 있습니다. 몇 가지 예제를 살펴보겠습니다.

Ambient 모듈 (Ambient Modules)

Node.js에서는 대부분의 작업은 하나 이상의 모듈을 로드하여 수행합니다. 최상위-레벨의 내보내기 선언으로 각 모듈을 `.d.ts` 파일로 정의할 수 있지만, 더 큰 하나의 `.d.ts` 파일로 모듈들을 작성하는 것이 더 편리합니다. 이를 위해, ambient 네임스페이스와 유사한 구조를 사용하지만, 나중에 import 할 수 있는 인용된 모듈 이름과 `module` 키워드를 사용합니다. 예를 들면:

node.d.ts (간단한 발췌)

```
declare module "url" {
    export interface Url {
        protocol?: string;
        hostname?: string;
        pathname?: string;
    }

    export function parse(urlStr: string, parseQueryString?, slashesDenoteHost?): Url;
}

declare module "path" {
    export function normalize(p: string): string;
    export function join(...paths: any[]): string;
    export var sep: string;
}
```

이제 `/// <reference> node.d.ts`를 수행한 다음, `import url = require("url");` 또는 `import * as URL from "url"`을 사용하여 모듈을 로드할 수 있습니다.

```
/// <reference path="node.d.ts"/>
import * as URL from "url";
let myUrl = URL.parse("http://www.typescriptlang.org");
```

속기 ambient 모듈 (Shorthand ambient modules)

새로운 모듈을 사용하기 전에 선언을 작성하지 않는 경우, 속기 선언(shorthand declaration)을 사용하여 빠르게 시작할 수 있습니다.

declarations.d.ts

```
declare module "hot-new-module";
```

속기 모듈로부터 모든 imports는 `any` 타입을 가집니다.

```
import x, {y} from "hot-new-module";
x(y);
```

와일드카드 모듈 선언 (Wildcard module declarations)

SystemJS나 AMD와 같은 모듈 로더는 비-JavaScirpt 내용을 import 할 수 있습니다. 이 둘은 일반적으로 접두사 또는 접미사를 사용하여 특수한 로딩 의미를 표시합니다. 이러한 경우를 다루기 위해 와일드카드 모듈 선언을 사용할 수 있습니다.

```
declare module "*!text" {
    const content: string;
    export default content;
}
// 일부는 다른 방법으로 사용합니다.
declare module "json!*" {
    const value: any;
    export default value;
}
```

이제 `"*!text"` 나 `"json!*"` 와 일치하는 것들을 import 할 수 있습니다.

```
import fileContent from "./xyz.txt!text";
import data from "json!http://example.com/data.json";
console.log(data, fileContent);
```

UMD 모듈 (UMD modules)

일부 라이브러리는 많은 모듈 로더에서 사용되거나, 모듈 로딩 (전역 변수) 없이 사용되도록 설계되었습니다. 이를 UMD 모듈이라고 합니다. 이러한 라이브러리는 import나 전역 변수를 통해 접근할 수 있습니다. 예를 들면:

math-lib.d.ts

```
export function isPrime(x: number): boolean;
export as namespace mathLib;
```

라이브러리는 모듈 내에서 import로 사용할 수 있습니다:

```
import { isPrime } from "math-lib";
isPrime(2);
mathLib.isPrime(2); // 오류: 모듈 내부에서 전역 정의를 사용할 수 없습니다.
```

전역 변수로도 사용할 수 있지만, 스크립트 내에서만 사용할 수 있습니다. (스크립트는 imports나 exports가 없는 파일입니다.)

```
mathLib.isPrime(2);
```

모듈 구조화에 대한 지침 (Guidance for structuring modules)

가능한 최상위-레벨에 가깝게 export 하기 (Export as close to top-level as possible)

모듈의 사용자가 export 모듈을 사용할 때 가능한 마찰이 적어야 합니다. 중첩 수준을 과도하게 추가하면 다루기 힘들어지는 경향이 있으므로, 어떻게 구조를 구성할지 신중하게 생각해야 합니다.

모듈에서 네임스페이스를 export 하는 것은 너무 많은 중첩 레이어를 추가하는 예입니다. 네임스페이스는 때때로 용도가 있지만, 모듈을 사용할 때 추가적인 레벨의 간접 참조를 추가합니다. 이것은 사용자에게 금방 고통스러운 지점이 될 수 있고, 일반적으로 불필요합니다.

export 한 클래스의 정적 메서드에도 비슷한 문제가 있습니다 - 클래스 자체에 중첩 레이어가 추가됩니다. 표현이나 의도를 명확하게 유용한 방식으로 높이지 않는 한 간단하게 헬퍼 함수를 export 하는 것을 고려하세요.

단일 `class` 나 `function` 을 export 할 경우, `export default` 를 사용하세요 (If you're only exporting a single `class` or `function`, use `export default`)

"최상위-레벨에 가까운 export"가 모듈 사용자의 마찰을 줄여주는 것처럼, default export를 도입하는 것도 마찬가지입니다. 모듈의 주요 목적이 한 개의 특정 export를 저장하는 것이라면, default export로 export 하는 것을 고려하세요. 이렇게 하면 import 하기와 실제로 import를 사용하기가 더 쉬워집니다. 예를 들면:

MyClass.ts

```
export default class SomeType {
  constructor() { ... }
}
```

MyFunc.ts

```
export default function getThing() { return "thing"; }
```

Consumer.ts

```
import t from "./MyClass";
import f from "./MyFunc";
let x = new t();
console.log(f());
```

이것은 사용자에게 최적입니다. 타입에 원하는 이름(이 경우에는 `t`)을 지정할 수 있고 객체를 찾기 위해 과도한 점을 찍지 않아도 됩니다.

여러 객체를 export 하는 경우, 최상위-레벨에 두세요 (If you're exporting multiple objects, put them all at top-level)

MyThings.ts

```
export class SomeType { /* ... */ }
export function someFunc() { /* ... */ }
```

반대로 import 할 때:

import 한 이름을 명시적으로 나열 (Explicitly list imported names)

Consumer.ts

```
import { SomeType, someFunc } from "./MyThings";
let x = new SomeType();
let y = someFunc();
```

많은 것을 import 하는 경우, 네임스페이스 import 패턴을 사용하세요 (Use the namespace import pattern if you're importing a large number of things)

MyLargeModule.ts

```
export class Dog { ... }
export class Cat { ... }
export class Tree { ... }
export class Flower { ... }
```

Consumer.ts

```
import * as myLargeModule from "./MyLargeModule.ts";
let x = new myLargeModule.Dog();
```

상속을 위한 re-export 하기 (Re-export to extend)

종종 모듈의 기능을 확장해야 할 필요가 있습니다. 일반적인 JS 패턴은 JQuery 확장이 작동하는 방식과 유사하게 *확장 (extensions)*으로 기존의 객체를 보강하는 것입니다. 앞에서 언급했듯이 모듈은 전역 네임스페이스 객체와 같이 *병합 (merge)* 하지 않습니다. 여기서 추천하는 방법은 기존의 객체를 *변형하지 않고* 새로운 기능을 제공하는 개체를 export 하는 것입니다.

`Calculator.ts` 모듈에 정의된 간단한 계산기 구현을 생각해보세요. 이 모듈도 입력 문자열 목록을 전달하고 결과를 작성하여 계산기의 기능을 테스트할 수 있는 헬퍼 함수를 export 합니다.

Calculator.ts

```ts
export class Calculator {
    private current = 0;
    private memory = 0;
    private operator: string;

    protected processDigit(digit: string, currentValue: number) {
        if (digit >= "0" && digit <= "9") {
            return currentValue * 10 + (digit.charCodeAt(0) - "0".charCodeAt(0));
        }
    }

    protected processOperator(operator: string) {
        if (["+", "-", "*", "/"].indexOf(operator) >= 0) {
            return operator;
        }
    }

    protected evaluateOperator(operator: string, left: number, right: number): number {
        switch (this.operator) {
            case "+": return left + right;
            case "-": return left - right;
            case "*": return left * right;
            case "/": return left / right;
        }
    }

    private evaluate() {
        if (this.operator) {
            this.memory = this.evaluateOperator(this.operator, this.memory, this.current);
        }
        else {
            this.memory = this.current;
        }
        this.current = 0;
    }

    public handleChar(char: string) {
        if (char === "=") {
            this.evaluate();
            return;
        }
        else {
            let value = this.processDigit(char, this.current);
            if (value !== undefined) {
                this.current = value;
                return;
            }
            else {
                let value = this.processOperator(char);
                if (value !== undefined) {
                    this.evaluate();
                    this.operator = value;
                    return;
                }
            }
        }
        throw new Error(`Unsupported input: '${char}'`);
    }

    public getResult() {
        return this.memory;
    }
}

export function test(c: Calculator, input: string) {
    for (let i = 0; i < input.length; i++) {
        c.handleChar(input[i]);
    }

    console.log(`result of '${input}' is '${c.getResult()}'`);
}
```

노출된 `test` 함수를 사용하는 간단한 계산기 테스트입니다.

TestCalculator.ts

```
import { Calculator, test } from "./Calculator";


let c = new Calculator();
test(c, "1+2*33/11="); // 9 출력
```

10이 아닌 숫자를 입력받을 수 있도록 이것을 상속하여 `ProgrammerCalculator.ts`을 만들어보겠습니다.

ProgrammerCalculator.ts

```
import { Calculator } from "./Calculator";

class ProgrammerCalculator extends Calculator {
    static digits = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "A", "B", "C", "D", "E", "F"];

    constructor(public base: number) {
        super();
        const maxBase = ProgrammerCalculator.digits.length;
        if (base <= 0 || base > maxBase) {
            throw new Error(`base has to be within 0 to ${maxBase} inclusive.`);
        }
    }

    protected processDigit(digit: string, currentValue: number) {
        if (ProgrammerCalculator.digits.indexOf(digit) >= 0) {
            return currentValue * this.base + ProgrammerCalculator.digits.indexOf(digit);
        }
    }
}

// 새로 상속된 calculator를 Calculator로 export 하기
export { ProgrammerCalculator as Calculator };

// 또한 헬퍼 함수도 export 하기
export { test } from "./Calculator";
```

새로운 `ProgrammerCalculator` 모듈은 `Calculator` 모듈과 유사한 API 형태를 export 하지만, 원래 모듈의 객체를 보강하지는 않습니다. 다음은 ProgrammerCalculator 클래스에 대한 테스트입니다:

TestProgrammerCalculator.ts

```
import { Calculator, test } from "./ProgrammerCalculator";

let c = new Calculator(2);
test(c, "001+010="); // 3 출력
```

모듈에서 네임스페이스를 사용하지 마세요 (Do not use namespaces in modules)

모듈 기반 구성을 처음 적용할 때, 일반적으로 추가적인 네임스페이스 계층에서 exports를 래핑 하는 경향이 있습니다. 모듈에는 자체 스코프가 있으며, export된 선언만 모듈 외부에서 볼 수 있습니다. 이를 염두에 두고 네임스페이스는 모듈을 다룰 때 거의 값을 변경하지 않습니다.

구성 전면에서 네임스페이스는 논리적으로 관련된 개체와 타입을 전역 스코프로 그룹화하는데 편리합니다. 예를 들어, C#의 경우, System.Collections에서 모든 컬렉션 타입을 찾을 수 있습니다. 타입을 계층적 네임스페이스로 구성하여 해당 타입의 사용자에게 "발견"할 수 있는 좋은 경험을 제공합니다. 반면, 모듈은 이미 파일 시스템에 반드시 존재합니다. 경로와 파일 이름으로 해석하기 위해서, 논리적 구성 체계를 사용할 수 있습니다. 리스트 모듈이 있는 /collections/generic/ 폴더를 사용할 수 있습니다.

네임스페이스는 전역 스코프에서 네이밍 충돌을 피하기 위해 중요합니다. 예를 들어, `My.Application.Customer.AddForm`과 `My.Application.Order.AddForm` -- 두 타입의 이름은 같지만 다른 네임스페이스를 가지고 있습니다. 그러나 이것은 모듈에서 문제가 되지 않습니다. 모듈 내에서 두 개의 객체가 같은 이름을 가질만한 이유는 없습니다. 사용 측면에서 특정 모듈의

사용자는 모듈을 참조하는데 사용할 이름을 선택하므로 우연한 이름 충돌은 불가능합니다.