단일 파일 검사기 (Validators in a single file)

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

let lettersRegexp = /^[A-Za-z]+$/;
let numberRegexp = /^[0-9]+$/;

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

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

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

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

// 각 문자열이 각 검사기를 통과했는지 표시
for (let s of strings) {
    for (let name in validators) {
        let isMatch = validators[name].isAcceptable(s);
        console.log(`'${ s }' ${ isMatch ? "matches" : "does not match" } '${ name }'.`);
    }
}
```

네임스페이스 적용하기 (Namespacing)

더 많은 검사기를 추가하게 되면, 타입을 추적하고 다른 객체와 이름 충돌을 방지하기 위해 일종의 구조 체계가 필요합니다. 전역 네임스페이스에 다른 이름을 많이 넣는 대신, 객체들을 하나의 네임스페이스로 감싸겠습니다.

이 예에서는 모든 검사기 관련 개체를 `Validation` 이라는 하나의 네임스페이스로 옮기겠습니다.
여기서 인터페이스 및 클래스가 네임스페이스 외부에서도 접근 가능하도록 선언부에 `export` 를 붙입니다.
반면, 변수 `letterRegexp` 와 `numberRegexp` 는 구현 세부 사항이므로 외부로 내보내지 않아 네임스페이스 외부 코드에서 접근할 수 없습니다. 파일 하단의 테스트 코드에서, 네임스페이스 외부에서 사용될 때 타입의 이름을 검증해야 합니다 (예: `Validation.LetterOnlyValidator`).

네임스페이스화된 검사기 (Namespaced Validators)

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

    const lettersRegexp = /^[A-Za-z]+$/;
    const numberRegexp = /^[0-9]+$/;

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

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

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

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

// 각 문자열이 각 검사기를 통과했는지 표시
for (let s of strings) {
    for (let name in validators) {
        console.log(`"${ s }" - ${ validators[name].isAcceptable(s) ? "matches" : "does not match'
    }
}
```

파일 간 분할 (Splitting Across Files)

애플리케이션 규모가 커지면, 코드를 여러 파일로 분할해야 유지 보수가 용이합니다.

다중 파일 네임스페이스 (Multi-file namespaces)

여기서 `Validation` 네임스페이스를 여러 파일로 분할합니다. 파일이 분리되어 있어도 같은 네임스페이스에 기여할 수 있고 마치 한 곳에서 정의된 것처럼 사용할 수 있습니다. 파일 간 의존성이 존재하므로, 참조 태그를 추가하여 컴파일러에게 파일 간의 관계를 알립니다. 그 외에 테스트 코드는 변경되지 않았습니다.

Validation.ts

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

LettersOnlyValidators.ts

```
/// <reference path="Validation.ts" />
namespace Validation {
    const lettersRegexp = /^[A-Za-z]+$/;
    export class LettersOnlyValidator implements StringValidator {
        isAcceptable(s: string) {
            return lettersRegexp.test(s);
        }
    }
}
```

ZipCodeValidators.ts

```
/// <reference path="Validation.ts" />
namespace Validation {
    const numberRegexp = /^[0-9]+$/;
    export class ZipCodeValidator implements StringValidator {
        isAcceptable(s: string) {
            return s.length === 5 && numberRegexp.test(s);
        }
    }
}
```

Test.ts

```
/// <reference path="Validation.ts" />
/// <reference path="LettersOnlyValidator.ts" />
/// <reference path="ZipCodeValidator.ts" />

// Some samples to try
let strings = ["Hello", "98052", "101"];

// Validators to use
let validators: { [s: string]: Validation.StringValidator; } = {};
validators["ZIP code"] = new Validation.ZipCodeValidator();
validators["Letters only"] = new Validation.LettersOnlyValidator();

// Show whether each string passed each validator
for (let s of strings) {
    for (let name in validators) {
        console.log(`"${ s }" - ${ validators[name].isAcceptable(s) ? "matches" : "does not match'
    }
}
```

파일이 여러 개 있으면 컴파일된 코드가 모두 로드되는지 확인해야 합니다. 이를 수행하는 두 가지 방법이 있습니다.

먼저, 모든 입력 파일을 하나의 JavaScript 출력 파일로 컴파일하기 위해 `--outFile` 플래그를 사용하여 연결 출력(concatenated output)을 사용할 수 있습니다:

```
tsc --outFile sample.js Test.ts
```

컴파일러는 파일에 있는 참조 태그를 기반으로 출력 파일을 자동으로 정렬합니다. 각 파일을 개별적으로 지정할 수도 있습니다:

```
tsc --outFile sample.js Validation.ts LettersOnlyValidator.ts ZipCodeValidator.ts Test.ts
```

또는 파일별 컴파일 (기본값)을 사용하여 각 입력 파일을 하나의 JavaScript 파일로 생성할 수 있습니다. 여러 JS 파일이 생성되는 경우, 웹 페이지에서 생성된 개별 파일을 적절한 순서로 로드하기 위해 `<script>` 태그를 사용해야 합니다. 예를 들어:

MyTestPage.html (인용)

```
    <script src="Validation.js" type="text/javascript" />
    <script src="LettersOnlyValidator.js" type="text/javascript" />
    <script src="ZipCodeValidator.js" type="text/javascript" />
    <script src="Test.js" type="text/javascript" />
```

별칭 (Aliases)

네임스페이스 작업을 단순화할 수 있는 또 다른 방법은 일반적으로 사용되는 객체의 이름을 더 짧게 만들기 위해 `import q = x.y.z`를 사용하는 것입니다. 모듈을 로드하는 데 사용되는 `import x = require ("name")` 구문과 혼동하지 않기 위해, 이 구문은 단순히 특정 심벌에 별칭을 생성합니다. 이러한 종류의 가져오기(일반적으로 별칭이라고 함)는 모듈 가져오기에서 생성된 객체를 포함하여 모든 종류의 식별자에 대해 사용할 수 있습니다.

```
namespace Shapes {
    export namespace Polygons {
        export class Triangle { }
        export class Square { }
    }
}

import polygons = Shapes.Polygons;
let sq = new polygons.Square(); // 'new Shapes.Polygons.Square()'와 동일
```

`require` 키워드를 사용하지 않는다는 것을 명심하세요; 대신 가져오는 심벌은 정해진 이름으로 직접 할당합니다. `var`를 사용하는 것과 비슷하지만, 가져온 심벌의 타입 및 네임스페이스 의미에 대해서도 동작합니다. 특히, 값의 경우 `import`는 원래 심벌와 별개의 참조이므로 별칭 `var`에 대한 변경 내용은 원래 변수에 반영되지 않습니다.

다른 JavaScript 라이브러리로 작업하기 (Working with Other JavaScript Libraries)

TypeScript로 작성되지 않은 라이브러리의 형태를 설명하려면, 라이브러리가 외부에 제공하는 API를 선언해야 합니다. 대부분의 JavaScript 라이브러리는 소수의 최상위 객체만 노출하므로 네임스페이스를 사용하는 것이 좋습니다.

구현을 정의하지 않은 선언을 "ambient"라고 부릅니다. 일반적으로 이것은 `.d.ts` 파일에 정의되어 있습니다. C/C++에 익숙하다면 이를 `.h` 파일로 생각할 수 있습니다. 몇 가지 예를 살펴보겠습니다.

Ambient 네임스페이스 (Ambient Namespaces)

널리 사용되는 D3 라이브러리는 `d3` 이라는 전역 객체에서 기능을 정의합니다. 이 라이브러리는 `<script>` 태그를 통해 로드되므로(모듈 로더 대신) 형태를 정의하기 위해 선언할 때 네임스페이스를 사용합니다. TypeScript 컴파일러는 이 형태를 보기 위해, ambient 네임스페이스 선언을 사용합니다. 예를 들어 다음과 같이 작성할 수 있습니다:

D3.d.ts (간단한 인용)

```
declare namespace D3 {
    export interface Selectors {
        select: {
            (selector: string): Selection;
            (element: EventTarget): Selection;
        };
    }

    export interface Event {
        x: number;
        y: number;
    }

    export interface Base extends Selectors {
        event: Event;
    }
}

declare var d3: D3.Base;
```