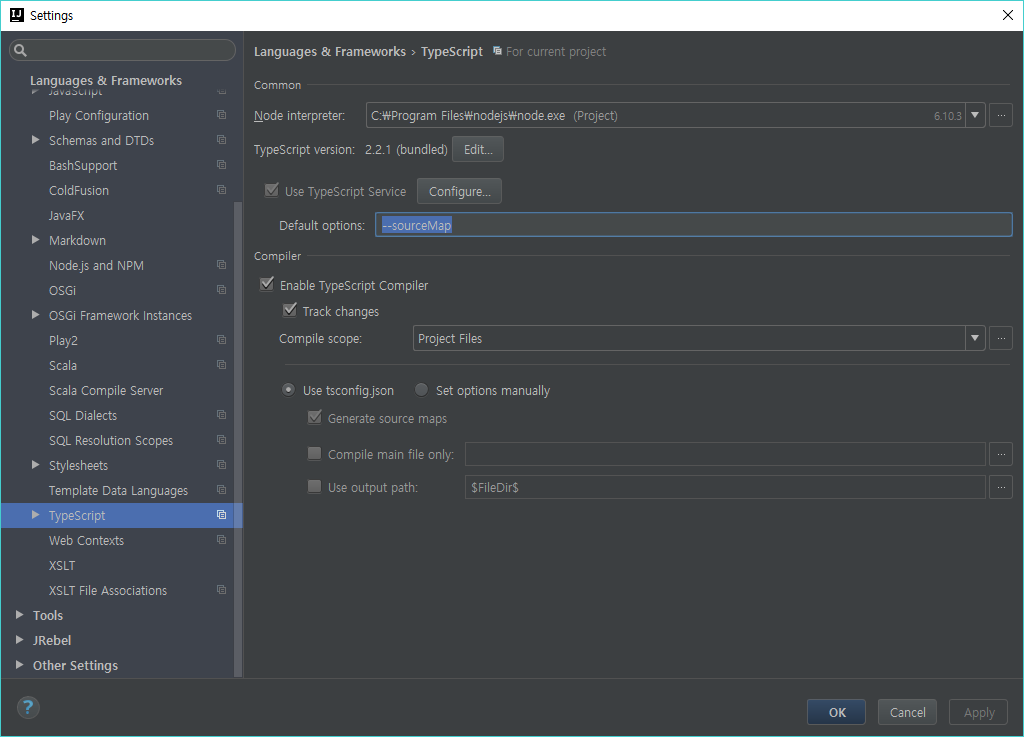
http://slides.com/woongjae/deck-8#/

<http://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html>

tsc -w --sourceMap



|  |
| --- |
| npm install -g typescript  또는 와치 있는거 하려면  npm install -g tsc-watch --save-dev  tsc-watch server.ts --outDir ./dist --onSuccess "node ./dist/server.ts"  https://www.npmjs.com/package/tsc-watch |

gulp를 이용하자

|  |
| --- |
| <https://www.typescriptlang.org/docs/handbook/gulp.html>  mkdir src  mkdir dist  npm init  npm install -g gulp  npm install -g gulp-cli  npm install -g --save-dev typescript gulp gulp-typescript |
| tsconfig.json:  {  "files": [  "src/main.ts"  ],  "compilerOptions": {  "noImplicitAny": true,  "target": "es5"  }  } |
| gulpfile.js:  **var** gulp = require("gulp");  **var** ts = require("gulp-typescript");  **var** tsProject = ts.createProject("tsconfig.json");  gulp.task("default", **function** () {  **return** tsProject.src()  .pipe(tsProject())  .js.pipe(gulp.dest("dist"));  }); |
| gulp  node dist/main.js |

모듈 연동

|  |
| --- |
| src/greet.ts:  **export** **function** **sayHello**(name: string) {  **return** `Hello from ${name}`;  }  Now change the code in src/main.ts to import sayHello from greet.ts:  **import** { sayHello } from "./greet";  console.log(sayHello("TypeScript")); |

|  |
| --- |
| **function** greeter(person) {  **return** "Hello, " + person; }  **var** user = "Jane User";  document.body.innerHTML = greeter(user); |
| **tsc init.ts 또는 tsc-watch ./init.ts**  **function** greeter(person) {  **return** "Hello, " + person; } **var** user = "Jane User"; document.body.innerHTML = greeter(user); |

interface 인터페이스

|  |
| --- |
| **interface** Person {  firstName: **string**;  lastName: **string**; }  **function** greeter(person: Person) {  **return** "Hello, " + person.firstName + " " + person.lastName; }  **var** user = { firstName: "Jane", lastName: "User" };  console.log( greeter(user) ); |
| **function** greeter(person) {  **return** "Hello, " + person.firstName + " " + person.lastName; } **var** user = { firstName: "Jane", lastName: "User" }; console.log(greeter(user)); |
| **Hello, Jane User** |

var , let , const

|  |  |  |
| --- | --- | --- |
| var 는 상당히 너그러운 친구입니다.   |  |  | | --- | --- | | 1  2 | var foo = 'bar1';  var foo = 'bar2'; |   같은 이름의 변수를 두번 선언하고 있습니다. foo라는 변수에는 ‘bar1′ 이라는 문자열이 할당되었다가 다음 선언에서 ‘bar2’라는 변수가 할당됩니다. 별다른 에러도 발생시키지 않습니다 |

let

|  |  |  |
| --- | --- | --- |
| 하지만 let과 const는 엄격합니다.   |  |  | | --- | --- | | 1  2  3  4 | let foo = 'bar1';  let foo = 'bar2';    // ERROR: Uncaught SyntaxError: Identifier 'foo' has already been declared |   앞서 선언한 변수를 다시 선언하게 되면 칼 같은 오류를 발생시킵니다. 규모가 큰 코드에서 버그를 방지 할 수 있는 매우 바람직한 특징입니다. |

유효범위를 가진다 let, const

|  |  |  |
| --- | --- | --- |
| 하지만 let과 const Block-scope 라고 합니다. 유효 범위가 블록, 즉 {}로 감싸지 범위라는 이야기 입니다.   |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9 | let foo = 'bar1';  console.log(foo); // bar1    if (true) {    let foo = 'bar2';    console.log(foo) // bar2  }    console.log(foo); // bar1 |   위 코드에서는 var를 사용한 경우와는 달리 if문 밖의 foo와 if문 안의 foo는 서로 다른 변수 입니다. 따라서 중복 선언으로 인한 에러도 발생하지 않으며, if문 안쪽에서 선언한 foo의 경우는 if문이 닫히는 시점에서 유효범위가 끝납니다. |

|  |  |  |
| --- | --- | --- |
| 그런데 여기서 의문이 생깁니다. if문 안에서 foo를 먼저 호출한 다음 let으로 foo를 다시 선언하게 되면 어떤일이 발생할까요? 일단 if 문에서 단순히 foo를 호출만 해보면   |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10 | let foo = 'bar1';  console.log(foo); // bar1    if (true) {    console.log(foo) // bar1    foo = 'bar2';    console.log(foo) // bar2  }    console.log(foo); // bar2 |   정상적으로 호출도되고 값의 변경에도 아무 문제가 없습니다. 그런 foo 호출 이후에 let으로 foo를 선언해 보겠습니 |

### **var VS let, const**

* var
  + ES5
  + 변수의 유효 범위 : 함수 스코프
  + 호이스팅 (O)
  + 재선언 가능
* let, const
  + ES6
  + 변수의 유효 범위 : 블록 스코프 (친숙)
  + 호이스팅 (X)
  + 재선언 불가
* var 말고 let, const

Basic Types기본타입

|  |
| --- |
| <http://yjoo00.tistory.com/56>  [Basic Types]  **1. Boolean**  true/false 값을 가지고있다.  **let** isDone: boolean = false;  /\* test.ts 파일에 위와같이 입력 후 tsc를 사용해 컴파일하면 아래와같은 test.js가 만들어진다. \*/  **var** test = **false**;  /\* typescript에서 선언한 boolean 타입이 사라지고 var로만 표시되는 것을 확인할 수 있다. \*/  **2. Number**  float형을 사용하며 2진수, 8진수, 10진수, 16진수 또한 지원한다.  **let** decimal: number = 6;  **let** hex: number = 0xf00d;  **let** binary: number = 0b1010;  **let** octal: number = 0o744;  /\* test.ts 파일에 위와같이 입력 후 tsc를 사용해 컴파일하면 아래와같은 test.js가 만들어진다. \*/  **var** decimal = 6;  **var** hex = 0xf00d;  **var** binary = 10;  **var** octal = 484;  /\* typescript에서 선언한 number 타입이 사라지고 var로만 표시되는 것을 확인할 수 있다. \*/  **3. String**  JavaScript처럼 TypeScript 또한 큰따옴표(") 또는 작은따옴표(')룰 문자열 양끝에 사용한다.  **let** color: string = "blue";  color = 'red';  /\* test.ts 파일에 위와같이 입력 후 tsc를 사용해 컴파일하면 아래와같은 test.js가 만들어진다. \*/  **var** color = "**blue";**  color = "**red"**;  /\* typescript에서 선언한 string 타입이 사라지고 var로만 표시되는 것을 확인할 수 있다. \*/  또한 문자열 주변을 backquote(`)로 감싼 후 입력 받은 변수를 ${exptr} 다음과 같이 표현하여 여럿 라인을  이을수있다.  **let** fullName: string = `Bob Bobbington`;  **let** age: number = 37;  **let** sentence: string = `Hello, my name is ${ fullName }.  I'll be ${ age + 1 } years old next month.`  /\* test.ts 파일에 위와같이 입력 후 tsc를 사용해 컴파일하면 아래와같은 test.js가 만들어진다. \*/  **var** fullName = "Bob Bobbington";  **var** age = 37;  **va**r sentence =**"**Hello, my name is**" +** fullName**+ "**.\n\nI'll be**" +**(age + 1) **+ "** years old next month**."**;  /\* backqutoe(`)가 큰따옴표(")로 변경되며 이어지는 문자들이 +로 변경된다. \*/  **4. Array**  JavaScript와 마찬가지로 TypeScript에서도 Array 사용이 가능하다.  Array type은 두가지 방법으로 사용 가능한다.  첫번째는 literal array을 이용한 방식이다.  **let** list: number[] = [1, 2, 3];  두번째는 generic array를 이용한 방식이다.  **let** list: Array<number> = [1, 2, 3];  두 방식 모두 tsc를 이용해 컴파일하면 다음과 같은 test.js가 만들어진다.  **var** list = [1, 2, 3];  **5. Tuple**  Tuple Type은 요소로 가질 수 있는 Type이 여러가지로 정의될수있다라는 것을 제외하면 Array와 비슷하다.  // Declare a tuple type  **let** x: [string, number];  첫번째 요소엔 String type, 두번째 요소엔 number type을 가진 Array x를 정의한 것이다.  // Declare a tuple type  **let** x: [string, number];  // Initialize it  x = ["hello", 10]; // OK  // Initialize it incorrectly  x = [10, "hello"]; // Error  // Initialize it correctly  x = [10, "hello", 10]; // OK'number' can be assigned to 'string | number'  // Initialize it incorrectly  x = [10, "hello", flase]; // Error'boolean' isn't 'string | number'  첫번째 요소, 두번째 요소에 처음 정의된 type과 다른 type을 입력 시 Error를 리턴한다.  그러면?! 정의되지않은 세번째 요소에 값을 넣으면?  결과는 첫번째, 두번째 요소에 사용된 type이면 문제없지만 사용되지않은 새로운 type이면 문제가 발생한다.  **6. Enum**  C#과같이 enum은 numeric value를 지정해줄수있는 친근한 방법이다.  **enum** Color {Red, Green, Blue};  **let** c: Color = Color.Green;  /\* test.ts 파일에 위와같이 입력 후 tsc를 사용해 컴파일하면 아래와같은 test.js가 만들어진다. \*/ var Color; (function (Color) { Color[Color["Red"] = 0] = "Red"; Color[Color["Green"] = 1] = "Green"; Color[Color["Blue"] = 2] = "Blue"; })(Color || (Color = {})); ; var c = Color.Green; /\* javascript로 변환되니 지금까지와는 다르게 코드가 생성되었다.  이는 javascript에서는 사용하지않는 enum을 enum과 같이 동작하게끔 만들어서이다. \*/  **7. Any**  JavaScript와 같이 하나의 변수에 어떠한 타입을 입력해도 오류가 나지않는다.  **let** notSure: any = 4;  notSure = "maybe a string instead";  notSure = false; // okay, definitely a boolean  /\* any type사용 시 number, string, boolean type 모두 ok! \*/  **8. Void**  이 리턴하는 값이 없는 함수일 경우 리턴값으로 void type을 명시할수있다.  **function** **warnUser**(): **void** {  alert("This is my warning message"); return null;  }  NULL & Undefined type과 마찬가지로 null과 undefined밖에 할당할수밖에 없어 유용하게 사용되지는 않는다.  **9. Null and Undefined**  // Not much else we can assign to these variables!  **let** u: undefined = undefined;  **let** n: null = null;  **10. union**  '|'를 사용하여 type을 조합할수 있다.  **let** a: string | number = "aaaa"; /\* 해당변수는 string type과 number type 둘 다 사용한다. \*/  **a = 4; /\* number 사용 OK \*/a = true; /\* boolean 사용 ERROR \*/**      **11. never**  함수를 활용하는 경우 예외가 아니고는 리턴을 하지않음을 보장해야할 경우가 있다.  예를 들면, 항상 에러를 호출하는 함수이거나 무한루프가 포함된 함수이다.  // Function returning never must have unreachable end point  **function** **error**(message: string): **never** {  **throw** **new** Error(message);  }  // Inferred return type is never  **function** **fail**() {  **return** error("Something failed");  }  // Function returning never must have unreachable end point  **function** **infiniteLoop**(): **never** {  **while** (true) {  }  }    **12. Type assertions**  형변환 타입으로 다음과 같은 방법으로 가능하다.  **let** someValue: any = "this is a string";  **let** strLength: number = (<**string**>someValue).length;  또는 as - 구문으로 가능하다.  **let** someValue: any = "this is a string";  **let** strLength: number = (someValue as string).length; |

[Variable Declarations]

**val & let 차이**

**var**는 JavaScript에서 정의할때 사용하는 키워드이다.

var는 아래 예제와같은 사용성(?)을 가지고있다.

1. A 함수에서 사용하는 변수 a를  B함수에서 호출할 수 있다.

**function** **f**() {

**var** a = 10;

**return** **function** **g**() {

**var** b = a + 1;

**return** b;

}

}

**var** g = f();

g(); // returns '11'

2.  정의되지않을 가능성이있는 변수 사용이 가능하다.

**function** **f**(shouldInitialize: boolean) {

**if** (shouldInitialize) {

**var** x = 10;

}

**return** x;

}

f(true); // returns '10'

f(false); // returns 'undefined'

3.  중복된 변수 선언이 가능하다. (for문안에 i를 선언한 후 다시 for문안에 i를 선언한다.)

**function** **sumMatrix**(matrix: number[][]) {

**var** sum = 0;

**for** (**var** i = 0; i < matrix.length; i++) {

**var** currentRow = matrix[i];

**for** (**var** i = 0; i < currentRow.length; i++) {

sum += currentRow[i];

}

}

**return** sum;

}

4. var의 경우 새로운 scope없이 한 변수에 인덱스가 추가되며 바인딩 되었고, 결국 setTimeout에 순차적으로 추가한 함수들이 같은 scope var를 참조하면서(clousre되면서) 예상과 드른결과값이 도출된다.

**for** (**var** i = 0; i < 10; i++) {

setTimeout(**function**() { console.log(i); }, 100 \* i);

}  
  
///// 결과는 10 10 10 10 10 10 10 10 10 10 이다.

위와같은 var의 몇가지 문제점이 왜 let을 사용해야되는지를 알려준다.

let은 다음과 같이 var의 문제점에서 해결해준다.

1. 정의되지않은 변수사용시 ERROR발생.

**function** **f**(input: boolean) {

**let** a = 100;

**if** (input) {

// Still okay to reference 'a'

**let** b = a + 1;

**return** b;

}

// Error: 'b' doesn't exist here

**return** b;

}

2. 중복선언 ERROR.

**let** x = 10;

**let** x = 20; // error: can't re-declare 'x' in the same scope

3.  block-scope의 변수는 다른 변수로 인지.

**function** **f**(condition, x) {

**if** (condition) {

**let** x = 100;

**return** x;

}

**return** x;

}

f(false, 0); // returns '0'

f(true, 0); // returns '100'

**let & const 차이**

위에서 var와 let의 차이를 대략적으로 보았다면, 과연 let과 const는 어떠한 차이가 있을까?

결론부터 이야기하면 **const는 불변 변수, let은 가변 변수**를 선언할때 쓰인다.

**const let** x = 10;

x = 20; // error

[interface]

1. type check를 위한 interface

**function** **printLabel**(labelledObj: { label: string }) {

console.log(labelledObj.label);

}

**let** myObj = {size: 10, label: "Size 10 Object"};

printLabel(myObj);

myObj는 size, label의 프로퍼티를 가지고있지만 printLabel의 파라미터는 오직 label 프로퍼티만 요청하고있다.

이는 컴파일러가 적어도 하나의 type만 확인하면 된다는것을 알려준다.

위의 예제를 interface를 사용하도록 변경하면 다음과 같다.

**interface** LabelledValue {

label: string;

}

**function** **printLabel**(labelledObj: LabelledValue) {

console.log(labelledObj.label);

}

**let** myObj = {size: 10, label: "Size 10 Object"};

printLabel(myObj);

interface LabelledValue에서 label 프로퍼티를 보여주고있다.

이는 printLabel에게 오직 string type label 프로퍼티만 충족시켜주면 충분하다고 말하고있는 것이다.

**2. optional 프로퍼티**

interface의 모든 프로퍼티가 요구되어지는 것은 아니다.

**interface** SquareConfig {

color?: string;

width?: number;

}

**function** **createSquare**(config: SquareConfig): {color: string; area: number} {

**let** newSquare = {color: "white", area: 100};

**if** (config.color) {

newSquare.color = config.color;

}

**if** (config.width) {

newSquare.area = config.width \* config.width;

}

**return** newSquare;

}

**let** mySquare = createSquare({color: "black"});

interface optional 프로퍼티는 식별자 뒤에 ?를 붙여서 사용하며 선택적으로 구현하는 프로퍼티이다.

위의 코드에서 볼수있듯이, 인터페이스를 구현하는 함수에서 optional 프로퍼티를 가지고있지 않아도 에러가 발생하지않는다.

3. Readonly 프로퍼티

typescript에서는 이름에서 유추가능하듯이 읽기만 가능한 readonly 프로퍼티를 선언할수있다.

**interface** Point {

readonly x: number;

readonly y: number;

}  
  
**let** p1: Point = { x: 10, y: 20 };

p1.x = 5; // error!

위의 예제와같이 x, y를 readonly 프로퍼티를 이용해 선언했기때문에 x값 변경 요청시 에러가 발생한다.

그럼, readonly 프로퍼티는 const와의 차이가 없는걸까? 라는 궁금증이 생긴다.

얼핏보면 똑같은 행위를하는듯 보이는 이 readonly 프로퍼티와 const의 가장 쉬운 차이점은 const는 변수에 사용되며 readonly는 프로퍼티에 사용된다.

뿐만 아니라 아래와같은 간단한 에제로 const와 readonly 차이를 알수있다.

**let** a: number[] = [1, 2, 3, 4];

**let** ro: ReadonlyArray<number> = a;  
ro[0] = 12; // error! ro = [1, 2, 3];  
const ro2 = a;ro2[0] = 12;ro2 = [1, 2, 3]; // error!

const도 상수화, readonly도 상수화일텐데 왜 결과값은 다를까?

이유는 readonly는 내가 가리키는 데이터에대한 상수화이고 const는 자기 자싲의 상수화이기 때문이다.

참고! ro와 ro2의 값은 copy가 아닌 reference된 값이다.

4. Excess Property Checks

**interface** SquareConfig {

color?: string;

width?: number;

}

**function** **createSquare**(config: SquareConfig): { color: string; area: number } { return ;

// ...

}

/\* TEST A\*/

**let** mySquare = createSquare({ colour: "red", width: 100 }); // ERROR  
/\* TEST B \*/**let** test = { colour: "red", width: 100 };**let** mySquare = createSquare(test); // Success

얼핏보면 TEST A는 존재하지않은 colour를 인자로 사용하여 발생한 ERROR로 보인다.

하지만 TEST B에서도 똑같이 존재하지않은 변수 colour를 인자로 넘기지만 TEST B에서는 ERROR가 발생하지않는다.

TEST A와 TEST B의 차이는 무엇일까?

바로 변수 test에다 colour를 사용여부이다.

TEST A와 같이 함수에 직접적으로 선언되지않은 colour를 사용할경우 해당 변수는 명백히 잘못사용된 인자가 되지만 TEST B와 같이 test 변수에 선언된 변수는 어디에선가 쓰일수도있는 변수이기때문에 ERROR가 발생하지않는다.

5. 함수타입

interface를 사용하여 함수의 파라미터, 리턴값의 type을 지정할수있다.

함수 인자의 이름은 변경될수 있으며 '()' 괄호를 사용해 선언하며 return값은 ':'으로 선언한다.

**interface** SearchFunc {

(source: string, subString: string): boolean;

}

**let** mySearch: SearchFunc;

mySearch = **function**(source: string, subString: string) {

**let** result = source.search(subString);

**if** (result == -1) {

**return** false;

}

**else** {

**return** true;

}

}

한번 지정한 함수 type은 다른 interface로도 사용가능하다.

위의 코드에서는 SearchFunc 함수 type을 mySearch에서 사용되는것을 보여준다.

이후에 나오는 class type, hybred type 등등은 class chapter를 정리 한 후 진행하도록 한다.

[Classes]

1. class (클래스)

간단한 class기반의 예제를 살펴보자.

**class** Greeter {

greeting: string;

**constructor**(message: string) {

**this**.greeting = message;

}

greet() {

**return** "Hello, " + **this**.greeting;

}

}

**let** greeter = **new** Greeter("world");

위 코드에서는 3가지 멤버(**greeting**이라고 불리는 property, **constructor**, **greet()** 메소드)를 가지고있는 Greeter Class를 정의하고있다.

참고로 constructor(생성자) 메소드는 class로 생성된 객체를 초기화하기위한 특수 메소드이다.

2. Inheritance (상속)

 상속 예제를 살펴보자

**class** Animal {

name: string;

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

move(distanceInMeters: number = 0) {

console.log(`${**this**.name} moved ${distanceInMeters}m.`);

}

}

**class** Snake extends Animal {

**constructor**(name: string) { **super**(name); }

move(distanceInMeters = 5) {

console.log("Slithering...");

**super**.move(distanceInMeters);

}

}

**class** Horse extends Animal {

**constructor**(name: string) { **super**(name); }

move(distanceInMeters = 45) {

console.log("Galloping...");

**super**.move(distanceInMeters);

}

}

**let** sam = **new** Snake("Sammy the Python");

**let** tom: Animal = **new** Horse("Tommy the Palomino");

sam.move();

tom.move(34);

extends를 사용하여 부모 class로 부터 특정 기능을 상속받을수있다.

위의 예제의 부모 class는 Animal이 될것이고 자식 class는 Snake와 Horse가 될것이다.

부모 class의 constructor 함수는 super 키워드를 통해 호출할수있으며,

Snake class, Horse class 둘 다 Animal class의 constructor 함수를 호출한것을 확인할수 있다.

따라서 sam.move()는 Snake class의 move함수에서 super.move를 통해 부모 class인 Animal class의 move함수로 이동할것이고

tom.move(34)는 Horse class의 move함수에서 super.move를 통해 부모 class인 Animal class의 move함수로 이동할것이다.

3. Public, private, and protected modifiers

접근제한자 public, private, protected에대해 알아보도록 한다.

예제를 확인하기전 각각의 기능을 간략하게 구분하면 다음과 같다.

**public**

어디에서나 해당 class, method 참조 가능하다.

**private**

자신을 포함한 class내에서만 참조 가능하다.

**protected**

자신과 상속받은 class내에서만 참조 가능하다.

**class** Animal {

**private** name: string;

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

}

**new** Animal("Cat").name; // Error: 'name' is private;

private로 선언한 name을 class 외부에서 호출시도하여 발생한 ERROR이며,

해당 ERROR를 해결하기위해서는 private를 public으로 수정하거나 제거하면된다.

( 기본적으로 접근 제한자를 사용하지 않는다면 모든 프러퍼티와 메소드는 public 이다.)

**class** Person {

**protected** name: string;

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

}

**class** Employee extends Person {

**private** department: string;

**constructor**(name: string, department: string) {

**super**(name);

**this**.department = department;

}

**public** getElevatorPitch() {

**return** `Hello, my name is ${**this**.name} and I work **in** ${**this**.department}.`;

}

}

**let** howard = **new** Employee("Howard", "Sales");

console.log(howard.getElevatorPitch());

console.log(howard.name); // error

protected로 선언한 name을 subclass인 Emloyee 에서는 사용가능하지만 그 외 외부에서 접근시 Error가 발생한다.

4. Accessors (접근자)

**class** Employee {

fullName: string;

}

**let** employee = **new** Employee();

employee.fullName = "Bob Smith";

**if** (employee.fullName) {

console.log(employee.fullName);

}

Typescript에서 지원하는 get 과 set 에대한 간단한 예제이다.

멤버 fullName에 값을 설정하고 리턴하는 역할을 한다.

5. Abstract Classes (추상화 Class)

abstract **class** Animal {

abstract makeSound(): void;

move(): void {

console.log("roaming the earth...");

}

}

추상화 class란 추상 method를 포함하고있는 class를 의미한다.

추상화 class 자체만으로는 인스턴트를 생성할수 없기 때문에 상속을 통해 자식 class에 의해서만 완성될수 있다.

Function

Function Type

|  |
| --- |
| **function** add(x: **number**, y: **number**): **number** {  **return** x + y; }  **let** myAdd = **function**(x: **number**, y: **number**): **number** { **return** x+y; }; |
| **let** myAdd: (x: **number**, y: **number**) => **number** =  **function**(x: **number**, y: **number**): **number** { **return** x+y; };  **var** myAdd = **function** (x, y) { **return** x + y; }; |
| **let** myAdd: (baseValue:**number**, increment:**number**) => **number** =  **function**(x: **number**, y: **number**): **number** { **return** x + y; }; |

Inferring the types 유형 추론

|  |
| --- |
| // myAdd has the full function type **let** myAdd1 = **function**(x: **number**, y: **number**): **number** { **return** x + y; };  // The parameters 'x' and 'y' have the type number **let** myAdd2: (baseValue:**number**, increment:**number**) => **number** =  **function**(x, y) { **return** x + y; }; |
| // myAdd has the full function type **var** myAdd1 = **function** (x, y) { **return** x + y; }; // The parameters 'x' and 'y' have the type number **var** myAdd2 = **function** (x, y) { **return** x + y; }; |

Optional and Default Parameters 기본 매개변수

|  |
| --- |
| **function** buildName(firstName: **string**, lastName = "Smith") {  **return** firstName + " " + lastName; }  **let** result1 = buildName("Bob"); // works correctly now, returns "Bob Smith" **let** result2 = buildName("Bob", undefined); // still works, also returns "Bob Smith" **let** result3 = buildName("Bob", "Adams", "Sr."); // error, too many parameters **let** result4 = buildName("Bob", "Adams"); // ah, just right  console.log(result1) console.log(result2) console.log(result3) console.log(result4) |
| **function** buildName(firstName, lastName) {  **if** (lastName === **void** 0) { lastName = "Smith"; }  **return** firstName + " " + lastName; } **var** result1 = buildName("Bob"); // works correctly now, returns "Bob Smith" **var** result2 = buildName("Bob", undefined); // still works, also returns "Bob Smith" **var** result3 = buildName("Bob", "Adams", "Sr."); // error, too many parameters **var** result4 = buildName("Bob", "Adams"); // ah, just right console.log(result1); console.log(result2); console.log(result3); console.log(result4);  Bob Smith  Bob Smith  Bob Adams  Bob Adams |

|  |
| --- |
| **function** **buildName**(firstName = "Will", lastName: string) {  **return** firstName + " " + lastName;  }  **let** result1 = buildName("Bob"); // error, too few parameters  **let** result2 = buildName("Bob", "Adams", "Sr."); // error, too many parameters  **let** result3 = buildName("Bob", "Adams"); // okay and returns "Bob Adams"  **let** result4 = buildName(undefined, "Adams"); // okay and returns "Will Adams" |

여러파라미터

|  |
| --- |
| **function** buildName(firstName: **string**, ...restOfName: **string**[]) {  **return** firstName + " " + restOfName.join(" "); }  **let** employeeName = buildName("Joseph", "Samuel", "Lucas", "MacKinzie"); |
| **function** buildName(firstName) {  **var** restOfName = [];  **for** (**var** \_i = 1; \_i < arguments.length; \_i++) {  restOfName[\_i - 1] = arguments[\_i];  }  **return** firstName + " " + restOfName.join(" "); } **var** employeeName = buildName("Joseph", "Samuel", "Lucas", "MacKinzie"); |

|  |
| --- |
| **function** buildName(firstName: **string**, ...restOfName: **string**[]) {  **return** firstName + " " + restOfName.join(" "); }  **let** buildNameFun: (fname: **string**, ...rest: **string**[]) => **string** = buildName; **function** buildName(firstName) {  **var** restOfName = [];  **for** (**var** \_i = 1; \_i < arguments.length; \_i++) {  restOfName[\_i - 1] = arguments[\_i];  }  **return** firstName + " " + restOfName.join(" "); } **var** buildNameFun = buildName; |
|  |

자바스크립트의 this 해결 하는 옛날방법.

오류나는것

|  |
| --- |
| **var** deck = {  suits: ["hearts", "spades", "clubs", "diamonds"],  cards: Array(52),  createCardPicker: **function** () {  **return function** () {  **var** pickedCard = Math.floor(Math.random() \* 52);  **var** pickedSuit = Math.floor(pickedCard / 13);  **return** { suit: this.suits[pickedSuit], card: pickedCard % 13 };  };  } }; **var** cardPicker = deck.createCardPicker(); **var** pickedCard = cardPicker(); alert("card: " + pickedCard.card + " of " + pickedCard.suit);  함수안쪽에 있는 this를 하게되면 Call 하는 당사자의 this를 가르키기때문에 this.suits를 찾을수 없다  해결하려면 .call 이나 apply를 해서 호출하는 쪽에서 코드레벨 변경해야되거나 아래처럼 해야한다. |

해결방법

|  |
| --- |
| **var** deck = {  suits: ["hearts", "spades", "clubs", "diamonds"],  cards: Array(52),  createCardPicker: **function** () {  **var \_this = this;**  // NOTE: the line below is now an arrow function, allowing us to capture 'this' right here  **return function** () {  **var** pickedCard = Math.floor(Math.random() \* 52);  **var** pickedSuit = Math.floor(pickedCard / 13);  **return** { suit: \_this.suits[pickedSuit], card: pickedCard % 13 };  };  } }; **var** cardPicker = deck.createCardPicker(); **var** pickedCard = cardPicker(); alert("card: " + pickedCard.card + " of " + pickedCard.suit);  위쪽에 임시적인 this를 만든 변수 값을 지정한후 처리하면 정상처리된다. |

타입스크립트로는 이런식으로 처리가능하다

|  |
| --- |
| **interface** Card {  suit: **string**;  card: **number**; } **interface** Deck {  suits: **string**[];  cards: **number**[];  **createCardPicker(this: Deck): () => Card;** } **let** deck: Deck = {  suits: ["hearts", "spades", "clubs", "diamonds"],  cards: Array(52),  // NOTE: The function now explicitly specifies that its callee must be of type Deck  createCardPicker: **function**(**this: Deck**) {  **return** () => {  **let** pickedCard = Math.floor(Math.random() \* 52);  **let** pickedSuit = Math.floor(pickedCard / 13);   **return** {suit: **this**.suits[pickedSuit], card: pickedCard % 13};  }  } }  **let** cardPicker = deck.createCardPicker(); **let** pickedCard = cardPicker();  alert("card: " + pickedCard.card + " of " + pickedCard.suit); |
| **var** deck = {  suits: ["hearts", "spades", "clubs", "diamonds"],  cards: Array(52),  // NOTE: The function now explicitly specifies that its callee must be of type Deck  createCardPicker: **function** () {  **var \_this = this;**  **return function** () {  **var** pickedCard = Math.floor(Math.random() \* 52);  **var** pickedSuit = Math.floor(pickedCard / 13);  **return** { suit: \_this.suits[pickedSuit], card: pickedCard % 13 };  };  } }; **var** cardPicker = deck.createCardPicker(); **var** pickedCard = cardPicker(); alert("card: " + pickedCard.card + " of " + pickedCard.suit); |
| **// 위처럼 변환된다.** |

구조분석

|  |
| --- |
| **let** myAdd: **(x: number, y: number) => number =**  **function** (x: **number**, y: **number**): **number** {  **return** x + y;  };  (x:..y:..) 나는 함수야  => 그런데 구현체는 이거야  number 리턴값은 이거구  = function... 구현부분은 여기 야 |
|  |
| **let** myAdd: **number =** (**function** (): **number** {  **return** 100; })();  let myAdd 변수명은 myAdd이야  : number 리턴값은 number야  = (function..{...})) 값은 이거야 |
|  |
| **let** x = () => ({name: "Alice"}); |
| **let** x = (a: number) => 0; |
| let x 변수명은 x야  = () 나는 함수고  => 리턴값은 이거야. |
| **다시보기 해보기—**  **interface** gg{  name:**string**; }  **let** gg2 : gg = (**function**(){  **return** {name:"ggg"}; })(); **let** gg3 : () => gg = **function**(){  **return** {name:"ggg"}; };  console.log(gg2); console.log(gg3); |
| **{ name: 'ggg' }**  **[Function: gg3]** |

타입스크립트와 타입스크립트가 아닌걸로 했을때

|  |
| --- |
| **let** myAdd: (x: **number**, y: **number**) => **number** =  **function** (x: **number**, y: **number**): **number** {  **return** "11" + x + y; **<-- 에러에러**  }; **let** myAdd3 = **function**(x: **number**, y: **number**){  **return** "11" +x+y; } |

Generics 제너릭

|  |  |
| --- | --- |
| **function** identity0(arg: **number**): **number** {  **return** arg; }  **function** identity1(arg: **any**): **any** {  **return** arg; }  **function** identity2<T>(arg: T): T {  **return** arg; }  **function** loggingIdentity1<T>(arg: T[]): T[] {  console.log(arg.length); // Array has a .length, so no more error  **return** arg; }  **function** loggingIdentity2<T>(arg: Array<T>): Array<T> {  console.log(arg.length); // Array has a .length, so no more error  **return** arg; } | **function** identity0(arg) {  **return** arg; } **function** identity1(arg) {  **return** arg; } **function** identity2(arg) {  **return** arg; } |

|  |
| --- |
| **function** identity2<T>(arg: T): T {  **return** arg; }  **let** output = identity2<**string**>("myString"); // type of output will be 'string' // let output = identity("myString");  **let** myIdentity1: <T>(arg: T) => T = identity2 **let** myIdentity2: {<T>(arg: T): T} = identity2;  console.log(myIdentity1<**any**>('44')) console.log(myIdentity2<**string**>("2")) |
| **function** identity2(arg) {  **return** arg; } **var** output = identity2("myString"); // type of output will be 'string' // let output = identity("myString"); **var** myIdentity1 = identity2; **var** myIdentity2 = identity2; console.log(myIdentity1('44')); console.log(myIdentity2("2")); |

제너릭을 이용한뒤 특정 제네릭으로 처리가능하도록 하기 연속기.

|  |
| --- |
| **interface** GenericIdentityFn<T> {  (arg: T): T; }  **function** identity<T>(arg: T): T {  **return** arg; }  **let** myIdentity: GenericIdentityFn<**number**> = identity; |

Generic Classes

|  |
| --- |
| **class** GenericNumber<T> {  zeroValue: T;  add: (x: T, y: T) => T; }  **let** myGenericNumber = **new** GenericNumber<**number**>(); myGenericNumber.zeroValue = 11; myGenericNumber.add = **function**(x, y) { **return** x + y; }; console.log(myGenericNumber.add(myGenericNumber.zeroValue,4444));    **let** stringNumeric = **new** GenericNumber<**string**>(); stringNumeric.zeroValue = "sss"; stringNumeric.add = **function**(x, y) { **return** x + y; }; console.log(stringNumeric.add(stringNumeric.zeroValue, "test"));  4455  ssstest |

Generic Constraints 제약

|  |
| --- |
| **interface** Lengthwise {  length: **number**; }  **function** loggingIdentity<T **extends** Lengthwise>(arg: T): T {  console.log(arg.length); // Now we know it has a .length property, so no more error  **return** arg; } |

제너릭 파라미터 제약

|  |
| --- |
| **function** getProperty<T, K **extends keyof** T>(obj: T, key: K) {  **return** obj[key]; }  **let** x = { a: 1, b: 2, c: 3, d: 4 };  getProperty(x, "a"); // okay **getProperty(x, "m"); // error: Argument of type 'm' isn't assignable to 'a' | 'b' | 'c' | 'd'.**  오류남 아래꺼 |

제너릭 클래스 제약

|  |
| --- |
| **function** create<T>(c: {**new**(): T; }): T {  **return new** c(); }   **class** BeeKeeper {  hasMask: **boolean**; }  **class** ZooKeeper {  nametag: **string**; }  **class** Animal {  numLegs: **number**; }  **class** Bee **extends** Animal {  keeper: BeeKeeper; }  **class** Lion **extends** Animal {  keeper: ZooKeeper; }  **function** createInstance<A **extends** Animal>(c: **new** () => A): A {  **return new** c(); }  createInstance(Lion).keeper.nametag; // typechecks! createInstance(Bee).keeper.hasMask; // typechecks! |

Enums 열거형

|  |
| --- |
| **enum** Direction {  *Up* = 1,  *Down*,  *Left*,  *Right* } |

|  |  |
| --- | --- |
| **enum** FileAccess {  // constant members  *None*,  *Read* = 1 << 1,  *Write* = 1 << 2,  *ReadWrite* = *Read* | *Write*,  // computed member  *G* = "123".length }  console.log(FileAccess.*None*) console.log(FileAccess.*Read*) console.log(FileAccess.*Write*) console.log(FileAccess.*ReadWrite*) console.log(FileAccess.*G*)  0  2  4  6  3 | **var** FileAccess; (**function** (FileAccess) {  // constant members  FileAccess[FileAccess["None"] = 0] = "None";  FileAccess[FileAccess["Read"] = 2] = "Read";  FileAccess[FileAccess["Write"] = 4] = "Write";  FileAccess[FileAccess["ReadWrite"] = 6] = "ReadWrite";  // computed member  FileAccess[FileAccess["G"] = "123".length] = "G"; })(FileAccess || (FileAccess = {})); console.log(FileAccess.None); console.log(FileAccess.Read); console.log(FileAccess.Write); console.log(FileAccess.ReadWrite); console.log(FileAccess.G); |

|  |
| --- |
| **enum** Enum {  *A* } **let** a = Enum.*A*; **let** nameOfA = Enum[Enum.*A*]; // "A" console.log(nameOfA)  A |

|  |
| --- |
| **const enum** Directions {  *Up*,  *Down*,  *Left*,  *Right* }  **let** directions = [Directions.*Up*, Directions.*Down*, Directions.*Left*, Directions.*Right*] |
| **var** directions = [0 /\* Up \*/, 1 /\* Down \*/, 2 /\* Left \*/, 3 /\* Right \*/]; |

|  |
| --- |
| **declare enum** Enum {  *A* = 1,  *B*,  *C* = 2 } |

Type Inference 유형추론

Type Compatibility 유형호환성

|  |
| --- |
| **interface** Named {  name: **string**; }  **let** x: Named; // y's inferred type is { name: string; location: string; } **let** y = { name: "Alice", location: "Seattle" }; x = y; |

error

|  |
| --- |
| **let** x = (a: **number**) => 0; **let** y = (b: **number**, s: **string**) => 0;  y = x; // OK x = y; // Error |

Function Parameter Bivariance 매개변수 이항

Optional Parameters and Rest Parameters 옵셔날 파라미터 리셋파라마터

|  |
| --- |
| **function** invokeLater(args: **any**[], callback: (...args: **any**[]) => **void**) {  /\* ... Invoke callback with 'args' ... \*/ }  // Unsound - invokeLater "might" provide any number of arguments invokeLater([1, 2], (x, y) => console.log(x + ", " + y));  // Confusing (x and y are actually required) and undiscoverable invokeLater([1, 2], (x?, y?) => console.log(x + ", " + y)); |

Advanced Types 고급유형

|  |
| --- |
| **function** extend<T, U>(first: T, second: U): T & U {  **let** result = <T & U>{};  **for** (**let** id **in** first) {  (<**any**>result)[id] = (<**any**>first)[id];  }  **for** (**let** id **in** second) {  **if** (!result.hasOwnProperty(id)) {  (<**any**>result)[id] = (<**any**>second)[id];  }  }  **return** result; }  **class** Person {  **constructor**(**public** name: **string**) { } } **interface** Loggable {  log(): **void**; } **class** ConsoleLogger **implements** Loggable {  log() {  // ...  } } **var** jim = extend(**new** Person("Jim"), **new** ConsoleLogger()); **var** n = jim.name; jim.log(); |
|  |

Union Types 유니온 타입.  
Nullable types

Type guards and type assertions

**function** **f**(sn: string | null): **string** {

**if** (sn == null) {

**return** "default";

}

**else** {

**return** sn;

}

}

**let** person: **string** | **number** = 0;  
person = 'Mark';  
  
**type** StringOrNumber = **string** | **number**;  
  
**let** another: StringOrNumber = 0;  
another = 'Anna';  
  
/\*  
  
 1. 유니온 타입은 A 도 가능하고 B 도 가능한 타입  
 2. 길게 쓰는걸 짧게  
  
 \*/

Aliasing Tuple

|  |
| --- |
| **let** person: [**string**, **number**] = ['Mark', 35];  **type** PersonTuple = [**string**, **number**];  **let** another: PersonTuple = ['Anna', 24];  /\*   1. 튜플 타입에 별칭을 줘서 여러군데서 사용할 수 있게 한다.   \*/ |

The null elimination is pretty obvious here, but you can use terser operators too:

**function** **f**(sn: string | null): **string** {

**return** sn || "default";

}

|  |
| --- |
| **function** broken(name: **string** | **null**): **string** {  **function** postfix(epithet: **string**) {  **return** name.charAt(0) + '. the ' + epithet; // error, 'name' is possibly null  }  name = name || "Bob";  **return** postfix("great"); }  **function** fixed(name: **string** | **null**): **string** {  **function** postfix(epithet: **string**) {  **return** name!.charAt(0) + '. the ' + epithet; // ok  }  name = name || "Bob";  **return** postfix("great"); } |

Type Aliases

Symbols

Starting with ECMAScript 2015, symbol is a primitive data type, just like number and string.

symbol values are created by calling the Symbol constructor.

**let** sym1 = Symbol();

**let** sym2 = Symbol("key"); // optional string key

Symbols are immutable, and unique.

**let** sym2 = Symbol("key");

**let** sym3 = Symbol("key");

sym2 === sym3; // false, symbols are unique

Just like strings, symbols can be used as keys for object properties.

**let** sym = Symbol();

**let** obj = {

[sym]: "value"

};

console.log(obj[sym]); // "value"

Symbols can also be combined with computed property declarations to declare object properties and class members.

**const** getClassNameSymbol = Symbol();

**class** C {

[getClassNameSymbol](){

**return** "C";

}

}

**let** c = **new** C();

**let** className = c[getClassNameSymbol](); // "C"

# Well-known Symbols

In addition to user-defined symbols, there are well-known built-in symbols. Built-in symbols are used to represent internal language behaviors.

Here is a list of well-known symbols:

## Symbol.hasInstance

A method that determines if a constructor object recognizes an object as one of the constructor’s instances. Called by the semantics of the instanceof operator.

## Symbol.isConcatSpreadable

A Boolean value indicating that an object should be flatten to its array elements by Array.prototype.concat.

## Symbol.iterator

A method that returns the default iterator for an object. Called by the semantics of the for-of statement.

## Symbol.match

A regular expression method that matches the regular expression against a string. Called by the String.prototype.match method.

## Symbol.replace

A regular expression method that replaces matched substrings of a string. Called by the String.prototype.replacemethod.

## Symbol.search

A regular expression method that returns the index within a string that matches the regular expression. Called by the String.prototype.search method.

## Symbol.species

A function valued property that is the constructor function that is used to create derived objects.

## Symbol.split

A regular expression method that splits a string at the indices that match the regular expression. Called by the String.prototype.split method.

## Symbol.toPrimitive

A method that converts an object to a corresponding primitive value. Called by the ToPrimitive abstract operation.

## Symbol.toStringTag

A String value that is used in the creation of the default string description of an object. Called by the built-in method Object.prototype.toString.

## Symbol.unscopables

An Object whose own property names are property names that are excluded from the ‘with’ environment bindings of the associated objects.

Iterators and Generators 반복 및 생성기

|  |
| --- |
| **let** someArray = [1, "string", **false**];  **for** (**let** entry **of** someArray) {  console.log(entry); // 1, "string", false } |

Modules

<https://www.typescriptlang.org/docs/handbook/modules.html>

Export

|  |
| --- |
| **export interface** StringValidator {  isAcceptable(s: **string**): **boolean**; }  **export const** numberRegexp = /^[0-9]+$/;  **export class** ZipCodeValidator **implements** StringValidator {  isAcceptable(s: **string**) {  **return** s.length === 5 && numberRegexp.test(s);  } } |
| **export** \* **from** "./StringValidator"; // exports interface 'StringValidator' **export** \* **from** "./LettersOnlyValidator"; // exports class 'LettersOnlyValidator' **export** \* **from** "./ZipCodeValidator"; // exports class 'ZipCodeValidator' |

## Export statements

Export statements are handy when exports need to be renamed for consumers, so the above example can be written as:

**class** ZipCodeValidator **implements** StringValidator {

isAcceptable(s: string) {

**return** s.length === 5 && numberRegexp.test(s);

}

}

**export** { ZipCodeValidator };

**export** { ZipCodeValidator as mainValidator };

## Re-exports

Often modules extend other modules, and partially expose some of their features. A re-export does not import it locally, or introduce a local variable.

##### ***ParseIntBasedZipCodeValidator.ts***

**export** **class** ParseIntBasedZipCodeValidator {

isAcceptable(s: string) {

**return** s.length === 5 && parseInt(s).toString() === s;

}

}

// Export original validator but rename it

**export** {ZipCodeValidator as RegExpBasedZipCodeValidator} from "./ZipCodeValidator";

Optionally, a module can wrap one or more modules and combine all their exports using export \* from "module"syntax.

##### ***AllValidators.ts***

**export** \* from "./StringValidator"; // exports interface 'StringValidator'

**export** \* from "./LettersOnlyValidator"; // exports class 'LettersOnlyValidator'

**export** \* from "./ZipCodeValidator"; // exports class 'ZipCodeValidator'

import

|  |
| --- |
| **import** { ZipCodeValidator } **from** "./ZipCodeValidator";  **let** myValidator = **new** ZipCodeValidator(); |

# mport

Importing is just about as easy as exporting from a module. Importing an exported declaration is done through using one of the import forms below:

## Import a single export from a module

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

**let** myValidator = **new** ZipCodeValidator();

imports can also be renamed

**import** { ZipCodeValidator as ZCV } from "./ZipCodeValidator";

**let** myValidator = **new** ZCV();

## 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();

## Import a module for side-effects only

Though not recommended practice, some modules set up some global state that can be used by other modules. These modules may not have any exports, or the consumer is not interested in any of their exports. To import these modules, use:

**import** "./my-module.js";

# Default exports

Each module can optionally export a default export. Default exports are marked with the keyword default; and there can only be one default export per module. default exports are imported using a different import form.

default exports are really handy. For instance, a library like JQuery might have a default export of jQuery or $, which we’d probably also import under the name $ or jQuery.

##### ***JQuery.d.ts***

**declare** **let** $: JQuery;

**export** **default** $;

##### ***App.ts***

**import** $ from "JQuery";

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

Classes and function declarations can be authored directly as default exports. Default export class and function declaration names are optional.

##### ***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();

or

##### ***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"];

// Use function validate

strings.forEach(s => {

console.log(`"${s}" ${validate(s) ? " matches" : " does not match"}`);

});

default exports can also be just values:

##### ***OneTwoThree.ts***

**export** **default** "123";

##### ***Log.ts***

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

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

# export = and import = require()

Both CommonJS and AMD generally have the concept of an exports object which contains all exports from a module.

They also support replacing the exports object with a custom single object. Default exports are meant to act as a replacement for this behavior; however, the two are incompatible. TypeScript supports export = to model the traditional CommonJS and AMD workflow.

The export = syntax specifies a single object that is exported from the module. This can be a class, interface, namespace, function, or enum.

When importing a module using export =, TypeScript-specific import module = require("module") must be used to import the module.

##### ***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");

// Some samples to try

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

// Validators to use

**let** validator = **new** zip();

// Show whether each string passed each validator

strings.forEach(s => {

console.log(`"${ s }" - ${ validator.isAcceptable(s) ? "matches" : "does not match" }`);

});

# Code Generation for Modules

Depending on the module target specified during compilation, the compiler will generate appropriate code for Node.js ([CommonJS](http://wiki.commonjs.org/wiki/CommonJS)), require.js ([AMD](https://github.com/amdjs/amdjs-api/wiki/AMD)), isomorphic ([UMD](https://github.com/umdjs/umd)), [SystemJS](https://github.com/systemjs/systemjs), or [ECMAScript 2015 native modules](http://www.ecma-international.org/ecma-262/6.0/#sec-modules) (ES6) module-loading systems. For more information on what the define, require and register calls in the generated code do, consult the documentation for each module loader.

This simple example shows how the names used during importing and exporting get translated into the module loading code.

##### ***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;

# Simple Example

Below, we’ve consolidated the Validator implementations used in previous examples to only export a single named export from each module.

To compile, we must specify a module target on the command line. For Node.js, use --module commonjs; for require.js, use --module amd. For example:

tsc --module commonjs Test.ts

When compiled, each module will become a separate .js file. As with reference tags, the compiler will follow importstatements to compile dependent files.

##### ***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";

// Some samples to try

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

// Validators to use

**let** validators: { [s: string]: StringValidator; } = {};

validators["ZIP code"] = **new** ZipCodeValidator();

validators["Letters only"] = **new** LettersOnlyValidator();

// Show whether each string passed each validator

strings.forEach(s => {

**for** (**let** name **in** validators) {

console.log(`"${ s }" - ${ validators[name].isAcceptable(s) ? "matches" : "does not match" } ${ name }`);

}

});

# Optional Module Loading and Other Advanced Loading Scenarios

In some cases, you may want to only load a module under some conditions. In TypeScript, we can use the pattern shown below to implement this and other advanced loading scenarios to directly invoke the module loaders without losing type safety.

The compiler detects whether each module is used in the emitted JavaScript. If a module identifier is only ever used as part of a type annotations and never as an expression, then no require call is emitted for that module. This elision of unused references is a good performance optimization, and also allows for optional loading of those modules.

The core idea of the pattern is that the import id = require("...") statement gives us access to the types exposed by the module. The module loader is invoked (through require) dynamically, as shown in the if blocks below. This leverages the reference-elision optimization so that the module is only loaded when needed. For this pattern to work, it’s important that the symbol defined via an import is only used in type positions (i.e. never in a position that would be emitted into the JavaScript).

To maintain type safety, we can use the typeof keyword. The typeof keyword, when used in a type position, produces the type of a value, in this case the type of the module.

##### ***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("...")) { /\* ... \*/ }

}

##### ***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("...")) { /\* ... \*/ }

});

}

##### ***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("...")) { /\* ... \*/ }

});

}

# Working with Other JavaScript Libraries

To describe the shape of libraries not written in TypeScript, we need to declare the API that the library exposes.

We call declarations that don’t define an implementation “ambient”. Typically, these are defined in .d.ts files. If you’re familiar with C/C++, you can think of these as .h files. Let’s look at a few examples.

## Ambient Modules

In Node.js, most tasks are accomplished by loading one or more modules. We could define each module in its own .d.ts file with top-level export declarations, but it’s more convenient to write them as one larger .d.ts file. To do so, we use a construct similar to ambient namespaces, but we use the module keyword and the quoted name of the module which will be available to a later import. For example:

##### ***node.d.ts (simplified excerpt)***

**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;

}

Now we can /// <reference> node.d.ts and then load the modules using import url = require("url"); or import \* as URL from "url".

/// <reference path="node.d.ts"/>

**import** \* as URL from "url";

**let** myUrl = URL.parse("http://www.typescriptlang.org");

### *Shorthand ambient modules*

If you don’t want to take the time to write out declarations before using a new module, you can use a shorthand declaration to get started quickly.

##### ***declarations.d.ts***

**declare** **module** "hot-new-**module**";

All imports from a shorthand module will have the any type.

**import** x, {y} from "hot-new-module";

x(y);

### *Wildcard module declarations*

Some module loaders such as [SystemJS](https://github.com/systemjs/systemjs/blob/master/docs/overview.md#plugin-syntax) and [AMD](https://github.com/amdjs/amdjs-api/blob/master/LoaderPlugins.md) allow non-JavaScript content to be imported. These typically use a prefix or suffix to indicate the special loading semantics. Wildcard module declarations can be used to cover these cases.

**declare** **module** "\*!text" {

**const** content: string;

**export** **default** content;

}

// Some do it the other way around.

**declare** **module** "json!\*" {

**const** value: any;

**export** **default** value;

}

Now you can import things that match "\*!text" or "json!\*".

**import** fileContent from "./xyz.txt!text";

**import** data from "json!http://example.com/data.json";

console.log(data, fileContent);

### *UMD modules*

Some libraries are designed to be used in many module loaders, or with no module loading (global variables). These are known as [UMD](https://github.com/umdjs/umd) or [Isomorphic](http://isomorphic.net/) modules. These libraries can be accessed through either an import or a global variable. For example:

##### ***math-lib.d.ts***

**export** **const** isPrime(x: number): boolean;

**export** as namespace mathLib;

The library can then be used as an import within modules:

**import** { isPrime } from "math-lib";

isPrime(2);

mathLib.isPrime(2); // ERROR: can't use the global definition from inside a module

It can also be used as a global variable, but only inside of a script. (A script is a file with no imports or exports.)

mathLib.isPrime(2);

# Guidance for structuring modules

## Export as close to top-level as possible

Consumers of your module should have as little friction as possible when using things that you export. Adding too many levels of nesting tends to be cumbersome, so think carefully about how you want to structure things.

Exporting a namespace from your module is an example of adding too many layers of nesting. While namespaces sometime have their uses, they add an extra level of indirection when using modules. This can quickly becomes a pain point for users, and is usually unnecessary.

Static methods on an exported class have a similar problem - the class itself adds a layer of nesting. Unless it increases expressivity or intent in a clearly useful way, consider simply exporting a helper function.

### *If you’re only exporting a single class or function, use export default*

Just as “exporting near the top-level” reduces friction on your module’s consumers, so does introducing a default export. If a module’s primary purpose is to house one specific export, then you should consider exporting it as a default export. This makes both importing and actually using the import a little easier. For example:

#### *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());

This is optimal for consumers. They can name your type whatever they want (t in this case) and don’t have to do any excessive dotting to find your objects.

### *If you’re exporting multiple objects, put them all at top-level*

#### *MyThings.ts*

**export** **class** SomeType { /\* ... \*/ }

**export** **function** **someFunc**() { /\* ... \*/ }

Conversely when importing:

### *Explicitly list imported names*

#### *Consumer.ts*

**import** { SomeType, someFunc } from "./MyThings";

**let** x = **new** SomeType();

**let** y = someFunc();

### *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();

## Re-export to extend

Often you will need to extend functionality on a module. A common JS pattern is to augment the original object with extensions, similar to how JQuery extensions work. As we’ve mentioned before, modules do not merge like global namespace objects would. The recommended solution is to not mutate the original object, but rather export a new entity that provides the new functionality.

Consider a simple calculator implementation defined in module Calculator.ts. The module also exports a helper function to test the calculator functionality by passing a list of input strings and writing the result at the end.

#### *Calculator.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** handelChar(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.handelChar(input[i]);

}

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

}

Here is a simple test for the calculator using the exposed test function.

#### *TestCalculator.ts*

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

**let** c = **new** Calculator();

test(c, "1+2\*33/11="); // prints 9

Now to extend this to add support for input with numbers in bases other than 10, let’s create ProgrammerCalculator.ts

#### *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**();

**if** (base <= 0 || base > ProgrammerCalculator.digits.length) {

**throw** **new** Error("base has to be within 0 to 16 inclusive.");

}

}

**protected** processDigit(digit: string, currentValue: number) {

**if** (ProgrammerCalculator.digits.indexOf(digit) >= 0) {

**return** currentValue \* **this**.base + ProgrammerCalculator.digits.indexOf(digit);

}

}

}

// Export the new extended calculator as Calculator

**export** { ProgrammerCalculator as Calculator };

// Also, export the helper function

**export** { test } from "./Calculator";

The new module ProgrammerCalculator exports an API shape similar to that of the original Calculator module, but does not augment any objects in the original module. Here is a test for our ProgrammerCalculator class:

#### *TestProgrammerCalculator.ts*

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

**let** c = **new** Calculator(2);

test(c, "001+010="); // prints 3

## Do not use namespaces in modules

When first moving to a module-based organization, a common tendency is to wrap exports in an additional layer of namespaces. Modules have their own scope, and only exported declarations are visible from outside the module. With this in mind, namespace provide very little, if any, value when working with modules.

On the organization front, namespaces are handy for grouping together logically-related objects and types in the global scope. For example, in C#, you’re going to find all the collection types in System.Collections. By organizing our types into hierarchical namespaces, we provide a good “discovery” experience for users of those types. Modules, on the other hand, are already present in a file system, necessarily. We have to resolve them by path and filename, so there’s a logical organization scheme for us to use. We can have a /collections/generic/ folder with a list module in it.

Namespaces are important to avoid naming collisions in the global scope. For example, you might have My.Application.Customer.AddForm and My.Application.Order.AddForm – two types with the same name, but a different namespace. This, however, is not an issue with modules. Within a module, there’s no plausible reason to have two objects with the same name. From the consumption side, the consumer of any given module gets to pick the name that they will use to refer to the module, so accidental naming conflicts are impossible.

For more discussion about modules and namespaces see [Namespaces and Modules](https://www.typescriptlang.org/docs/handbook/namespaces-and-modules.html).

## Red Flags

All of the following are red flags for module structuring. Double-check that you’re not trying to namespace your external modules if any of these apply to your files:

* A file whose only top-level declaration is export namespace Foo { ... } (remove Foo and move everything ‘up’ a level)
* A file that has a single export class or export function (consider using export default)
* Multiple files that have the same export namespace Foo { at top-level (don’t think that these are going to combine into one Foo!)

Namespaces

**A note about terminology:** It’s important to note that in TypeScript 1.5, the nomenclature has changed. “Internal modules” are now “namespaces”. “External modules” are now simply “modules”, as to align with [ECMAScript 2015](http://www.ecma-international.org/ecma-262/6.0/)’s terminology, (namely that module X { is equivalent to the now-preferred namespace X {).

# Introduction

This post outlines the various ways to organize your code using namespaces (previously “internal modules”) in TypeScript. As we alluded in our note about terminology, “internal modules” are now referred to as “namespaces”. Additionally, anywhere the module keyword was used when declaring an internal module, the namespace keyword can and should be used instead. This avoids confusing new users by overloading them with similarly named terms.

# First steps

Let’s start with the program we’ll be using as our example throughout this page. We’ve written a small set of simplistic string validators, as you might write to check a user’s input on a form in a webpage or check the format of an externally-provided data file.

## 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);

}

}

// Some samples to try

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

// Validators to use

**let** validators: { [s: string]: StringValidator; } = {};

validators["ZIP code"] = **new** ZipCodeValidator();

validators["Letters only"] = **new** LettersOnlyValidator();

// Show whether each string passed each validator

**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 }'.`);

}

}

# Namespacing

As we add more validators, we’re going to want to have some kind of organization scheme so that we can keep track of our types and not worry about name collisions with other objects. Instead of putting lots of different names into the global namespace, let’s wrap up our objects into a namespace.

In this example, we’ll move all validator-related entities into a namespace called Validation. Because we want the interfaces and classes here to be visible outside the namespace, we preface them with export. Conversely, the variables lettersRegexp and numberRegexp are implementation details, so they are left unexported and will not be visible to code outside the namespace. In the test code at the bottom of the file, we now need to qualify the names of the types when used outside the namespace, e.g. Validation.LettersOnlyValidator.

## 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);

}

}

}

// 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" } ${ name }`);

}

}

# Splitting Across Files

As our application grows, we’ll want to split the code across multiple files to make it easier to maintain.

## Multi-file namespaces

Here, we’ll split our Validation namespace across many files. Even though the files are separate, they can each contribute to the same namespace and can be consumed as if they were all defined in one place. Because there are dependencies between files, we’ll add reference tags to tell the compiler about the relationships between the files. Our test code is otherwise unchanged.

##### ***Validation.ts***

namespace Validation {

**export** **interface** StringValidator {

isAcceptable(s: string): boolean;

}

}

##### ***LettersOnlyValidator.ts***

/// <reference path="Validation.ts" />

namespace Validation {

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

**export** **class** LettersOnlyValidator **implements** StringValidator {

isAcceptable(s: string) {

**return** lettersRegexp.test(s);

}

}

}

##### ***ZipCodeValidator.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 ") + name);

}

}

Once there are multiple files involved, we’ll need to make sure all of the compiled code gets loaded. There are two ways of doing this.

First, we can use concatenated output using the --outFile flag to compile all of the input files into a single JavaScript output file:

tsc --outFile sample.js Test.ts

The compiler will automatically order the output file based on the reference tags present in the files. You can also specify each file individually:

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

Alternatively, we can use per-file compilation (the default) to emit one JavaScript file for each input file. If multiple JS files get produced, we’ll need to use <script> tags on our webpage to load each emitted file in the appropriate order, for example:

##### ***MyTestPage.html (excerpt)***

<**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" />

# Aliases

Another way that you can simplify working with of namespaces is to use import q = x.y.z to create shorter names for commonly-used objects. Not to be confused with the import x = require("name") syntax used to load modules, this syntax simply creates an alias for the specified symbol. You can use these sorts of imports (commonly referred to as aliases) for any kind of identifier, including objects created from module imports.

namespace Shapes {

**export** namespace Polygons {

**export** **class** Triangle { }

**export** **class** Square { }

}

}

**import** polygons = Shapes.Polygons;

**let** sq = **new** polygons.Square(); // Same as 'new Shapes.Polygons.Square()'

Notice that we don’t use the require keyword; instead we assign directly from the qualified name of the symbol we’re importing. This is similar to using var, but also works on the type and namespace meanings of the imported symbol. Importantly, for values, import is a distinct reference from the original symbol, so changes to an aliased var will not be reflected in the original variable.

# Working with Other JavaScript Libraries

To describe the shape of libraries not written in TypeScript, we need to declare the API that the library exposes. Because most JavaScript libraries expose only a few top-level objects, namespaces are a good way to represent them.

We call declarations that don’t define an implementation “ambient”. Typically these are defined in .d.ts files. If you’re familiar with C/C++, you can think of these as .h files. Let’s look at a few examples.

## Ambient Namespaces

The popular library D3 defines its functionality in a global object called d3. Because this library is loaded through a <script> tag (instead of a module loader), its declaration uses namespaces to define its shape. For the TypeScript compiler to see this shape, we use an ambient namespace declaration. For example, we could begin writing it as follows:

##### ***D3.d.ts (simplified excerpt)***

**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;

# Namespaces and Modules

**A note about terminology:** It’s important to note that in TypeScript 1.5, the nomenclature has changed. “Internal modules” are now “namespaces”. “External modules” are now simply “modules”, as to align with [ECMAScript 2015](http://www.ecma-international.org/ecma-262/6.0/)’s terminology, (namely that module X { is equivalent to the now-preferred namespace X {).

# Introduction

This post outlines the various ways to organize your code using namespaces and modules in TypeScript. We’ll also go over some advanced topics of how to use namespaces and modules, and address some common pitfalls when using them in TypeScript.

See the [Modules](https://www.typescriptlang.org/docs/handbook/modules.html) documentation for more information about modules. See the [Namespaces](https://www.typescriptlang.org/docs/handbook/namespaces.html) documentation for more information about namespaces.

# Using Namespaces

Namespaces are simply named JavaScript objects in the global namespace. This makes namespaces a very simple construct to use. They can span multiple files, and can be concatenated using --outFile. Namespaces can be a good way to structure your code in a Web Application, with all dependencies included as <script> tags in your HTML page.

Just like all global namespace pollution, it can be hard to identify component dependencies, especially in a large application.

# Using Modules

Just like namespaces, modules can contain both code and declarations. The main difference is that modules declare their dependencies.

Modules also have a dependency on a module loader (such as CommonJs/Require.js). For a small JS application this might not be optimal, but for larger applications, the cost comes with long term modularity and maintainability benefits. Modules provide for better code reuse, stronger isolation and better tooling support for bundling.

It is also worth noting that, for Node.js applications, modules are the default and the recommended approach to structure your code.

Starting with ECMAScript 2015, modules are native part of the language, and should be supported by all compliant engine implementations. Thus, for new projects modules would be the recommended code organization mechanism.

# Pitfalls of Namespaces and Modules

In this section we’ll describe various common pitfalls in using namespaces and modules, and how to avoid them.

## /// <reference>-ing a module

A common mistake is to try to use the /// <reference ... /> syntax to refer to a module file, rather than using an import statement. To understand the distinction, we first need to understand how compiler can locate the type information for a module based on the path of an import (e.g. the ... in import x from "...";, import x = require("...");, etc.) path.

The compiler will try to find a .ts, .tsx, and then a .d.ts with the appropriate path. If a specific file could not be found, then the compiler will look for an ambient module declaration. Recall that these need to be declared in a .d.tsfile.

* myModules.d.ts
* // In a .d.ts file or .ts file that is not a module:
* **declare** **module** "SomeModule" {
* **export** **function** **fn**(): **string**;
* }
* myOtherModule.ts
* /// <reference path="myModules.d.ts" />
* **import** \* as m from "SomeModule";

The reference tag here allows us to locate the declaration file that contains the declaration for the ambient module. This is how the node.d.ts file that several of the TypeScript samples use is consumed.

## Needless Namespacing

If you’re converting a program from namespaces to modules, it can be easy to end up with a file that looks like this:

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

The top-level module here Shapes wraps up Triangle and Square for no reason. This is confusing and annoying for consumers of your module:

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

A key feature of modules in TypeScript is that two different modules will never contribute names to the same scope. Because the consumer of a module decides what name to assign it, there’s no need to proactively wrap up the exported symbols in a namespace.

To reiterate why you shouldn’t try to namespace your module contents, the general idea of namespacing is to provide logical grouping of constructs and to prevent name collisions. Because the module file itself is already a logical grouping, and its top-level name is defined by the code that imports it, it’s unnecessary to use an additional module layer for exported objects.

Here’s a revised example:

* shapes.ts
* **export** **class** Triangle { /\* ... \*/ }
* **export** **class** Square { /\* ... \*/ }
* shapeConsumer.ts
* **import** \* as shapes from "./shapes";
* **let** t = **new** shapes.Triangle();

## Trade-offs of Modules

Just as there is a one-to-one correspondence between JS files and modules, TypeScript has a one-to-one correspondence between module source files and their emitted JS files. One effect of this is that it’s not possible to concatenate multiple module source files depending on the module system you target. For instance, you can’t use the outFile option while targeting commonjs or umd, but with TypeScript 1.8 and later, [it’s possible](https://www.typescriptlang.org/docs/handbook/release-notes/typescript-1-8.html#concatenate-amd-and-system-modules-with---outfile) to use outFile when targeting amd or system.

# Module Resolution

This section assumes some basic knowledge about modules. Please see the [Modules](https://www.typescriptlang.org/docs/handbook/modules.html) documentation for more information.

Module resolution is the process the compiler uses to figure out what an import refers to. Consider an import statement like import { a } from "moduleA"; in order to check any use of a, the compiler needs to know exactly what it represents, and will need to check its definition moduleA.

At this point, the compiler will ask “what’s the shape of moduleA?” While this sounds straightforward, moduleA could be defined in one of your own .ts/.tsx files, or in a .d.ts that your code depends on.

First, the compiler will try to locate a file that represents the imported module. To do so the compiler follows one of two different strategies: [Classic](https://www.typescriptlang.org/docs/handbook/module-resolution.html#classic) or [Node](https://www.typescriptlang.org/docs/handbook/module-resolution.html#node). These strategies tell the compiler where to look for moduleA.

If that didn’t work and if the module name is non-relative (and in the case of "moduleA", it is), then the compiler will attempt to locate an [ambient module declaration](https://www.typescriptlang.org/docs/handbook/modules.html#ambient-modules). We’ll cover non-relative imports next.

Finally, if the compiler could not resolve the module, it will log an error. In this case, the error would be something like error TS2307: Cannot find module 'moduleA'.

## Relative vs. Non-relative module imports

Module imports are resolved differently based on whether the module reference is relative or non-relative.

A relative import is one that starts with /, ./ or ../. Some examples include:

* import Entry from "./components/Entry";
* import { DefaultHeaders } from "../constants/http";
* import "/mod";

Any other import is considered **non-relative**. Some examples include:

* import \* as $ from "jquery";
* import { Component } from "@angular/core";

A relative import is resolved relative to the importing file and cannot resolve to an ambient module declaration. You should use relative imports for your own modules that are guaranteed to maintain their relative location at runtime.

A non-relative import can be resolved relative to baseUrl, or through path mapping, which we’ll cover below. They can also resolve to [ambient module declarations](https://www.typescriptlang.org/docs/handbook/modules.html#ambient-modules). Use non-relative paths when importing any of your external dependencies.

## Module Resolution Strategies

There are two possible module resolution strategies: [Node](https://www.typescriptlang.org/docs/handbook/module-resolution.html#node) and [Classic](https://www.typescriptlang.org/docs/handbook/module-resolution.html#classic). You can use the --moduleResolution flag to specify the module resolution strategy. If not specified, the default is [Classic](https://www.typescriptlang.org/docs/handbook/module-resolution.html#classic) for --module AMD | System | ES2015 or [Node](https://www.typescriptlang.org/docs/handbook/module-resolution.html#node) otherwise.

### *Classic*

This used to be TypeScript’s default resolution strategy. Nowadays, this strategy is mainly present for backward compatibility.

A relative import will be resolved relative to the importing file. So import { b } from "./moduleB" in source file /root/src/folder/A.ts would result in the following lookups:

1. /root/src/folder/moduleB.ts
2. /root/src/folder/moduleB.d.ts

For non-relative module imports, however, the compiler walks up the directory tree starting with the directory containing the importing file, trying to locate a matching definition file.

For example:

A non-relative import to moduleB such as import { b } from "moduleB", in a source file /root/src/folder/A.ts, would result in attempting the following locations for locating "moduleB":

1. /root/src/folder/moduleB.ts
2. /root/src/folder/moduleB.d.ts
3. /root/src/moduleB.ts
4. /root/src/moduleB.d.ts
5. /root/moduleB.ts
6. /root/moduleB.d.ts
7. /moduleB.ts
8. /moduleB.d.ts

### *Node*

This resolution strategy attempts to mimic the [Node.js](https://nodejs.org/) module resolution mechanism at runtime. The full Node.js resolution algorithm is outlined in [Node.js module documentation](https://nodejs.org/api/modules.html#modules_all_together).

#### *How Node.js resolves modules*

To understand what steps the TS compiler will follow, it is important to shed some light on Node.js modules. Traditionally, imports in Node.js are performed by calling a function named require. The behavior Node.js takes will differ depending on if require is given a relative path or a non-relative path.

Relative paths are fairly straightforward. As an example, let’s consider a file located at /root/src/moduleA.js, which contains the import var x = require("./moduleB"); Node.js resolves that import in the following order:

1. As the file named /root/src/moduleB.js, if it exists.
2. As the folder /root/src/moduleB if it contains a file named package.json that specifies a "main" module. In our example, if Node.js found the file /root/src/moduleB/package.json containing { "main": "lib/mainModule.js" }, then Node.js will refer to /root/src/moduleB/lib/mainModule.js.
3. As the folder /root/src/moduleB if it contains a file named index.js. That file is implicitly considered that folder’s “main” module.

You can read more about this in Node.js documentation on [file modules](https://nodejs.org/api/modules.html#modules_file_modules) and [folder modules](https://nodejs.org/api/modules.html#modules_folders_as_modules).

However, resolution for a [non-relative module name](https://www.typescriptlang.org/docs/handbook/module-resolution.html#relative-vs-non-relative-module-imports) is performed differently. Node will look for your modules in special folders named node\_modules. A node\_modules folder can be on the same level as the current file, or higher up in the directory chain. Node will walk up the directory chain, looking through each node\_modules until it finds the module you tried to load.

Following up our example above, consider if /root/src/moduleA.js instead used a non-relative path and had the import var x = require("moduleB");. Node would then try to resolve moduleB to each of the locations until one worked.

1. /root/src/node\_modules/moduleB.js
2. /root/src/node\_modules/moduleB/package.json (if it specifies a "main" property)
3. /root/src/node\_modules/moduleB/index.js
4. /root/node\_modules/moduleB.js
5. /root/node\_modules/moduleB/package.json (if it specifies a "main" property)
6. /root/node\_modules/moduleB/index.js
7. /node\_modules/moduleB.js
8. /node\_modules/moduleB/package.json (if it specifies a "main" property)
9. /node\_modules/moduleB/index.js

Notice that Node.js jumped up a directory in steps (4) and (7).

You can read more about the process in Node.js documentation on [loading modules from node\_modules](https://nodejs.org/api/modules.html#modules_loading_from_node_modules_folders).

#### *How TypeScript resolves modules*

TypeScript will mimic the Node.js run-time resolution strategy in order to locate definition files for modules at compile-time. To accomplish this, TypeScript overlays the TypeScript source file extensions (.ts, .tsx, and .d.ts) over the Node’s resolution logic. TypeScript will also use a field in package.json named "typings" to mirror the purpose of "main" - the compiler will use it to find the “main” definition file to consult.

For example, an import statement like import { b } from "./moduleB" in /root/src/moduleA.ts would result in attempting the following locations for locating "./moduleB":

1. /root/src/moduleB.ts
2. /root/src/moduleB.tsx
3. /root/src/moduleB.d.ts
4. /root/src/moduleB/package.json (if it specifies a "typings" property)
5. /root/src/moduleB/index.ts
6. /root/src/moduleB/index.tsx
7. /root/src/moduleB/index.d.ts

Recall that Node.js looked for a file named moduleB.js, then an applicable package.json, and then for an index.js.

Similarly a non-relative import will follow the Node.js resolution logic, first looking up a file, then looking up an applicable folder. So import { b } from "moduleB" in source file /root/src/moduleA.ts would result in the following lookups:

1. /root/src/node\_modules/moduleB.ts
2. /root/src/node\_modules/moduleB.tsx
3. /root/src/node\_modules/moduleB.d.ts
4. /root/src/node\_modules/moduleB/package.json (if it specifies a "typings" property)
5. /root/src/node\_modules/moduleB/index.ts
6. /root/src/node\_modules/moduleB/index.tsx
7. /root/src/node\_modules/moduleB/index.d.ts
8. /root/node\_modules/moduleB.ts
9. /root/node\_modules/moduleB.tsx
10. /root/node\_modules/moduleB.d.ts
11. /root/node\_modules/moduleB/package.json (if it specifies a "typings" property)
12. /root/node\_modules/moduleB/index.ts
13. /root/node\_modules/moduleB/index.tsx
14. /root/node\_modules/moduleB/index.d.ts
15. /node\_modules/moduleB.ts
16. /node\_modules/moduleB.tsx
17. /node\_modules/moduleB.d.ts
18. /node\_modules/moduleB/package.json (if it specifies a "typings" property)
19. /node\_modules/moduleB/index.ts
20. /node\_modules/moduleB/index.tsx
21. /node\_modules/moduleB/index.d.ts

Don’t be intimidated by the number of steps here - TypeScript is still only jumping up directories twice at steps (8) and (15). This is really no more complex than what Node.js itself is doing.

## Additional module resolution flags

A project source layout sometimes does not match that of the output. Usually a set of build steps result in generating the final output. These include compiling .ts files into .js, and copying dependencies from different source locations to a single output location. The net result is that modules at runtime may have different names than the source files containing their definitions. Or module paths in the final output may not match their corresponding source file paths at compile time.

The TypeScript compiler has a set of additional flags to inform the compiler of transformations that are expected to happen to the sources to generate the final output.

It is important to note that the compiler will not perform any of these transformations; it just uses these pieces of information to guide the process of resolving a module import to its definition file.

### *Base URL*

Using a baseUrl is a common practice in applications using AMD module loaders where modules are “deployed” to a single folder at run-time. The sources of these modules can live in different directories, but a build script will put them all together.

Setting baseUrl informs the compiler where to find modules. All module imports with non-relative names are assumed to be relative to the baseUrl.

Value of baseUrl is determined as either:

* value of baseUrl command line argument (if given path is relative, it is computed based on current directory)
* value of baseUrl property in ‘tsconfig.json’ (if given path is relative, it is computed based on the location of ‘tsconfig.json’)

Note that relative module imports are not impacted by setting the baseUrl, as they are always resolved relative to their importing files.

You can find more documentation on baseUrl in [RequireJS](http://requirejs.org/docs/api.html#config-baseUrl) and [SystemJS](https://github.com/systemjs/systemjs/blob/master/docs/config-api.md#baseurl) documentation.

### *Path mapping*

Sometimes modules are not directly located under baseUrl. For instance, an import to a module "jquery" would be translated at runtime to "node\_modules/jquery/dist/jquery.slim.min.js". Loaders use a mapping configuration to map module names to files at run-time, see [RequireJs documentation](http://requirejs.org/docs/api.html#config-paths) and [SystemJS documentation](https://github.com/systemjs/systemjs/blob/master/docs/config-api.md#paths).

The TypeScript compiler supports the declaration of such mappings using "paths" property in tsconfig.json files. Here is an example for how to specify the "paths" property for jquery.

{

"compilerOptions": {

"baseUrl": ".", // This must be specified if "paths" is.

"paths": {

"jquery": ["node\_modules/jquery/dist/jquery"] // This mapping is relative to "baseUrl"

}

}

}

Please notice that "paths" are resolved relative to "baseUrl". When setting "baseUrl" to another value than ".", i.e. the directory of tsconfig.json, the mappings must be changed accordingly. Say, you set "baseUrl": "./src" in the above example, then jquery should be mapped to "../node\_modules/jquery/dist/jquery".

Using "paths" also allows for more sophisticated mappings including multiple fall back locations. Consider a project configuration where only some modules are available in one location, and the rest are in another. A build step would put them all together in one place. The project layout may look like:

projectRoot

├── folder1

│ ├── file1.ts (imports 'folder1/file2' and 'folder2/file3')

│ └── file2.ts

├── generated

│ ├── folder1

│ └── folder2

│ └── file3.ts

└── tsconfig.json

The corresponding tsconfig.json would look like:

{

"compilerOptions": {

"baseUrl": ".",

"paths": {

"\*": [

"\*",

"generated/\*"

]

}

}

}

This tells the compiler for any module import that matches the pattern "\*" (i.e. all values), to look in two locations:

1. "\*": meaning the same name unchanged, so map <moduleName> => <baseUrl>/<moduleName>
2. "generated/\*" meaning the module name with an appended prefix “generated”, so map <moduleName> => <baseUrl>/generated/<moduleName>

Following this logic, the compiler will attempt to resolve the two imports as such:

* import ‘folder1/file2’
  1. pattern ‘\*’ is matched and wildcard captures the whole module name
  2. try first substitution in the list: ‘\*’ -> folder1/file2
  3. result of substitution is non-relative name - combine it with baseUrl -> projectRoot/folder1/file2.ts.
  4. File exists. Done.
* import ‘folder2/file3’
  1. pattern ‘\*’ is matched and wildcard captures the whole module name
  2. try first substitution in the list: ‘\*’ -> folder2/file3
  3. result of substitution is non-relative name - combine it with baseUrl -> projectRoot/folder2/file3.ts.
  4. File does not exist, move to the second substitution
  5. second substitution ‘generated/\*’ -> generated/folder2/file3
  6. result of substitution is non-relative name - combine it with baseUrl -> projectRoot/generated/folder2/file3.ts.
  7. File exists. Done.

### *Virtual Directories with rootDirs*

Sometimes the project sources from multiple directories at compile time are all combined to generate a single output directory. This can be viewed as a set of source directories create a “virtual” directory.

Using ‘rootDirs’, you can inform the compiler of the roots making up this “virtual” directory; and thus the compiler can resolve relative modules imports within these “virtual” directories as if were merged together in one directory.

For example consider this project structure:

src

└── views

└── view1.ts (imports './template1')

└── view2.ts

generated

└── templates

└── views

└── template1.ts (imports './view2')

Files in src/views are user code for some UI controls. Files in generated/templates are UI template binding code auto-generated by a template generator as part of the build. A build step will copy the files in /src/views and /generated/templates/views to the same directory in the output. At run-time, a view can expect its template to exist next to it, and thus should import it using a relative name as "./template".

To specify this relationship to the compiler, use"rootDirs". "rootDirs" specify a list of roots whose contents are expected to merge at run-time. So following our example, the tsconfig.json file should look like:

{

"compilerOptions": {

"rootDirs": [

"src/views",

"generated/templates/views"

]

}

}

Every time the compiler sees a relative module import in a subfolder of one of the rootDirs, it will attempt to look for this import in each of the entries of rootDirs.

The flexibility of rootDirs is not limited to specifying a list of physical source directories that are logically merged. The supplied array may include any number of ad hoc, arbitrary directory names, regardless of whether they exist or not. This allows the compiler to capture sophisticated bundling and runtime features such as conditional inclusion and project specific loader plugins in a in a type safe way.

Consider an internationalization scenario where a build tool automatically generates locale specific bundles by interpolating a special path token, say #{locale}, as part of a relative module path such as ./#{locale}/messages. In this hypothetical setup the tool enumerates supported locales, mapping the abstracted path into ./zh/messages, ./de/messages, and so forth.

Assume that each of these modules exports an array of strings. For example ./zh/messages might contain:

**export** **default** [

"您好吗",

"很高兴认识你"

];

By leveraging rootDirs we can inform the compiler of this mapping and thereby allow it to safely resolve ./#{locale}/messages, even though the directory will never exist. For example, with the following tsconfig.json:

{

"compilerOptions": {

"rootDirs": [

"src/zh",

"src/de",

"src/#{locale}"

]

}

}

The compiler will now resolve import messages from './#{locale}/messages' to import messages from './zh/messages' for tooling purposes, allowing development in a locale agnostic manner without compromising design time support.

## Tracing module resolution

As discussed earlier, the compiler can visit files outside the current folder when resolving a module. This can be hard when diagnosing why a module is not resolved, or is resolved to an incorrect definition. Enabling the compiler module resolution tracing using --traceResolution provides insight in what happened during the module resolution process.

Let’s say we have a sample application that uses the typescript module. app.ts has an import like import \* as ts from "typescript".

│ tsconfig.json

├───node\_modules

│ └───typescript

│ └───lib

│ typescript.d.ts

└───src

app.ts

Invoking the compiler with --traceResolution

tsc --traceResolution

Results in an output such as:

======== Resolving module 'typescript' from 'src/app.ts'. ========

Module resolution kind is not specified, using 'NodeJs'.

Loading module 'typescript' from 'node\_modules' folder.

File 'src/node\_modules/typescript.ts' does not exist.

File 'src/node\_modules/typescript.tsx' does not exist.

File 'src/node\_modules/typescript.d.ts' does not exist.

File 'src/node\_modules/typescript/package.json' does not exist.

File 'node\_modules/typescript.ts' does not exist.

File 'node\_modules/typescript.tsx' does not exist.

File 'node\_modules/typescript.d.ts' does not exist.

Found 'package.json' at 'node\_modules/typescript/package.json'.

'package.json' has 'typings' field './lib/typescript.d.ts' that references 'node\_modules/typescript/lib/typescript.d.ts'.

File 'node\_modules/typescript/lib/typescript.d.ts' exist - use it as a module resolution result.

======== Module name 'typescript' was successfully resolved to 'node\_modules/typescript/lib/typescript.d.ts'. ========

#### *Things to look out for*

* Name and location of the import

======== Resolving module **‘typescript’** from **‘src/app.ts’**. ========

* The strategy the compiler is following

Module resolution kind is not specified, using **‘NodeJs’**.

* Loading of typings from npm packages

‘package.json’ has **‘typings’** field ‘./lib/typescript.d.ts’ that references ‘node\_modules/typescript/lib/typescript.d.ts’.

* Final result

======== Module name ‘typescript’ was **successfully resolved** to ‘node\_modules/typescript/lib/typescript.d.ts’. ========

## Using --noResolve

Normally the compiler will attempt to resolve all module imports before it starts the compilation process. Every time it successfully resolves an import to a file, the file is added to the set of files the compiler will process later on.

The --noResolve compiler options instructs the compiler not to “add” any files to the compilation that were not passed on the command line. It will still try to resolve the module to files, but if the file is not specified, it will not be included.

For instance:

#### *app.ts*

**import** \* as A from "moduleA" // OK, 'moduleA' passed on the command-line

**import** \* as B from "moduleB" // Error TS2307: Cannot find module 'moduleB'.

tsc app.ts moduleA.ts --noResolve

Compiling app.ts using --noResolve should result in:

* Correctly finding moduleA as it was passed on the command-line.
* Error for not finding moduleB as it was not passed.

## Common Questions

### *Why does a module in the exclude list still get picked up by the compiler?*

tsconfig.json turns a folder into a “project”. Without specifying any “exclude” or “files” entries, all files in the folder containing the tsconfig.json and all its sub-directories are included in your compilation. If you want to exclude some of the files use “exclude”, if you would rather specify all the files instead of letting the compiler look them up, use “files”.

That was tsconfig.json automatic inclusion. That does not embed module resolution as discussed above. If the compiler identified a file as a target of a module import, it will be included in the compilation regardless if it was excluded in the previous steps.

So to exclude a file from the compilation, you need to exclude it and **all** files that have an import or /// <reference path="..." /> directive to it.

# Declaration Merging

# Introduction

Some of the unique concepts in TypeScript describe the shape of JavaScript objects at the type level. One example that is especially unique to TypeScript is the concept of ‘declaration merging’. Understanding this concept will give you an advantage when working with existing JavaScript. It also opens the door to more advanced abstraction concepts.

For the purposes of this article, “declaration merging” means that the compiler merges two separate declarations declared with the same name into a single definition. This merged definition has the features of both of the original declarations. Any number of declarations can be merged; it’s not limited to just two declarations.

# Basic Concepts

In TypeScript, a declaration creates entities in at least one of three groups: namespace, type, or value. Namespace-creating declarations create a namespace, which contains names that are accessed using a dotted notation. Type-creating declarations do just that: they create a type that is visible with the declared shape and bound to the given name. Lastly, value-creating declarations create values that are visible in the output JavaScript.

| **Declaration Type** | **Namespace** | **Type** | **Value** |
| --- | --- | --- | --- |
| Namespace | X |  | X |
| Class |  | X | X |
| Enum |  | X | X |
| Interface |  | X |  |
| Type Alias |  | X |  |
| Function |  |  | X |
| Variable |  |  | X |

Understanding what is created with each declaration will help you understand what is merged when you perform a declaration merge.

# Merging Interfaces

The simplest, and perhaps most common, type of declaration merging is interface merging. At the most basic level, the merge mechanically joins the members of both declarations into a single interface with the same name.

**interface** Box {

height: number;

width: number;

}

**interface** Box {

scale: number;

}

**let** box: Box = {height: 5, width: 6, scale: 10};

Non-function members of the interfaces must be unique. The compiler will issue an error if the interfaces both declare a non-function member of the same name.

For function members, each function member of the same name is treated as describing an overload of the same function. Of note, too, is that in the case of interface A merging with later interface A, the second interface will have a higher precedence than the first.

That is, in the example:

**interface** Cloner {

clone(animal: Animal): Animal;

}

**interface** Cloner {

clone(animal: Sheep): Sheep;

}

**interface** Cloner {

clone(animal: Dog): Dog;

clone(animal: Cat): Cat;

}

The three interfaces will merge to create a single declaration as so:

**interface** Cloner {

clone(animal: Dog): Dog;

clone(animal: Cat): Cat;

clone(animal: Sheep): Sheep;

clone(animal: Animal): Animal;

}

Notice that the elements of each group maintains the same order, but the groups themselves are merged with later overload sets ordered first.

One exception to this rule is specialized signatures. If a signature has a parameter whose type is a single string literal type (e.g. not a union of string literals), then it will be bubbled toward the top of its merged overload list.

For instance, the following interfaces will merge together:

**interface** Document {

createElement(tagName: any): Element;

}

**interface** Document {

createElement(tagName: "div"): HTMLDivElement;

createElement(tagName: "span"): HTMLSpanElement;

}

**interface** Document {

createElement(tagName: string): HTMLElement;

createElement(tagName: "canvas"): HTMLCanvasElement;

}

The resulting merged declaration of Document will be the following:

**interface** Document {

createElement(tagName: "canvas"): HTMLCanvasElement;

createElement(tagName: "div"): HTMLDivElement;

createElement(tagName: "span"): HTMLSpanElement;

createElement(tagName: string): HTMLElement;

createElement(tagName: any): Element;

}

# Merging Namespaces

Similarly to interfaces, namespaces of the same name will also merge their members. Since namespaces create both a namespace and a value, we need to understand how both merge.

To merge the namespaces, type definitions from exported interfaces declared in each namespace are themselves merged, forming a single namespace with merged interface definitions inside.

To merge the namespace value, at each declaration site, if a namespace already exists with the given name, it is further extended by taking the existing namespace and adding the exported members of the second namespace to the first.

The declaration merge of Animals in this example:

namespace Animals {

**export** **class** Zebra { }

}

namespace Animals {

**export** **interface** Legged { numberOfLegs: number; }

**export** **class** Dog { }

}

is equivalent to:

namespace Animals {

**export** **interface** Legged { numberOfLegs: number; }

**export** **class** Zebra { }

**export** **class** Dog { }

}

This model of namespace merging is a helpful starting place, but we also need to understand what happens with non-exported members. Non-exported members are only visible in the original (un-merged) namespace. This means that after merging, merged members that came from other declarations cannot see non-exported members.

We can see this more clearly in this example:

namespace Animal {

**let** haveMuscles = true;

**export** **function** **animalsHaveMuscles**() {

**return** haveMuscles;

}

}

namespace Animal {

**export** **function** **doAnimalsHaveMuscles**() {

**return** haveMuscles; // <-- error, haveMuscles is not visible here

}

}

Because haveMuscles is not exported, only the animalsHaveMuscles function that shares the same un-merged namespace can see the symbol. The doAnimalsHaveMuscles function, even though it’s part of the merged Animalnamespace can not see this un-exported member.

# Merging Namespaces with Classes, Functions, and Enums

Namespaces are flexible enough to also merge with other types of declarations. To do so, the namespace declaration must follow the declaration it will merge with. The resulting declaration has properties of both declaration types. TypeScript uses this capability to model some of patterns in JavaScript as well as other programming languages.

## Merging Namespaces with Classes

This gives the user a way of describing inner classes.

**class** Album {

label: Album.AlbumLabel;

}

namespace Album {

**export** **class** AlbumLabel { }

}

The visibility rules for merged members is the same as described in the ‘Merging Namespaces’ section, so we must export the AlbumLabel class for the merged class to see it. The end result is a class managed inside of another class. You can also use namespaces to add more static members to an existing class.

In addition to the pattern of inner classes, you may also be familiar with JavaScript practice of creating a function and then extending the function further by adding properties onto the function. TypeScript uses declaration merging to build up definitions like this in a type-safe way.

**function** **buildLabel**(name: string): **string** {

**return** buildLabel.prefix + name + buildLabel.suffix;

}

namespace buildLabel {

**export** **let** suffix = "";

**export** **let** prefix = "Hello, ";

}

alert(buildLabel("Sam Smith"));

Similarly, namespaces can be used to extend enums with static members:

**enum** Color {

red = 1,

green = 2,

blue = 4

}

namespace Color {

**export** **function** **mixColor**(colorName: string) {

**if** (colorName == "yellow") {

**return** Color.red + Color.green;

}

**else** **if** (colorName == "white") {

**return** Color.red + Color.green + Color.blue;

}

**else** **if** (colorName == "magenta") {

**return** Color.red + Color.blue;

}

**else** **if** (colorName == "cyan") {

**return** Color.green + Color.blue;

}

}

}

# Disallowed Merges

Not all merges are allowed in TypeScript. Currently, classes can not merge with other classes or with variables. For information on mimicking class merging, see the [Mixins in TypeScript](https://www.typescriptlang.org/docs/handbook/mixins.html) section.

# Module Augmentation

Although JavaScript modules do not support merging, you can patch existing objects by importing and then updating them. Let’s look at a toy Observable example:

// observable.js

**export** **class** **Observable**<**T**> {

// ... implementation left as an exercise for the reader ...

}

// map.js

**import** { Observable } **from** "./observable";

Observable.prototype.map = **function** (f) {

// ... another exercise for the reader

}

This works fine in TypeScript too, but the compiler doesn’t know about Observable.prototype.map. You can use module augmentation to tell the compiler about it:

// observable.ts stays the same

// map.ts

**import** { Observable } from "./observable";

**declare** **module** "./observable" {

**interface** Observable<T> {

map<U>(f: (x: T) => U): Observable<U>;

}

}

Observable.prototype.map = **function** (f) {

// ... another exercise for the reader

}

// consumer.ts

**import** { Observable } from "./observable";

**import** "./map";

**let** o: Observable<number>;

o.map(x => x.toFixed());

The module name is resolved the same way as module specifiers in import/export. See [Modules](https://www.typescriptlang.org/docs/handbook/modules.html) for more information. Then the declarations in an augmentation are merged as if they were declared in the same file as the original. However, you can’t declare new top-level declarations in the augmentation – just patches to existing declarations.

## Global augmentation

You can also add declarations to the global scope from inside a module:

// observable.ts

**export** **class** Observable<T> {

// ... still no implementation ...

}

**declare** global {

**interface** Array<T> {

toObservable(): Observable<T>;

}

}

Array.prototype.toObservable = **function** () {

// ...

}

Global augmentations have the same behavior and limits as module augmentations.

# JSX

# Introduction

[JSX](https://facebook.github.io/jsx/) is an embeddable XML-like syntax. It is meant to be transformed into valid JavaScript, though the semantics of that transformation are implementation-specific. JSX came to popularity with the [React](http://facebook.github.io/react/) framework, but has since seen other applications as well. TypeScript supports embedding, type checking, and compiling JSX directly into JavaScript.

# Basic usage

In order to use JSX you must do two things.

1. Name your files with a .tsx extension
2. Enable the jsx option

TypeScript ships with three JSX modes: preserve, react, and react-native. These modes only affect the emit stage - type checking is unaffected. The preserve mode will keep the JSX as part of the output to be further consumed by another transform step (e.g. [Babel](https://babeljs.io/)). Additionally the output will have a .jsx file extension. The react mode will emit React.createElement, does not need to go through a JSX transformation before use, and the output will have a .jsfile extension. The react-native mode is the equivalent of preserve in that it keeps all JSX, but the output will instead have a .js file extension.

| **Mode** | **Input** | **Output** | **Output File Extension** |
| --- | --- | --- | --- |
| preserve | <div /> | <div /> | .jsx |
| react | <div /> | React.createElement("div") | .js |
| react-native | <div /> | <div /> | .js |

You can specify this mode using either the --jsx command line flag or the corresponding option in your [tsconfig.json](https://www.typescriptlang.org/docs/handbook/tsconfig-json.html)file.

Note: The identifier *React* is hard-coded, so you must make React available with an uppercase R.

# The as operator

Recall how to write a type assertion:

**var** foo = <**foo**>bar;

Here we are asserting the variable bar to have the type foo. Since TypeScript also uses angle brackets for type assertions, JSX’s syntax introduces certain parsing difficulties. As a result, TypeScript disallows angle bracket type assertions in .tsx files.

To make up for this loss of functionality in .tsx files, a new type assertion operator has been added: as. The above example can easily be rewritten with the as operator.

**var** foo = bar as foo;

The as operator is available in both .ts and .tsx files, and is identical in behavior to the other type assertion style.

# Type Checking

In order to understand type checking with JSX, you must first understand the difference between intrinsic elements and value-based elements. Given a JSX expression <expr />, expr may either refer to something intrinsic to the environment (e.g. a div or span in a DOM environment) or to a custom component that you’ve created. This is important for two reasons:

1. For React, intrinsic elements are emitted as strings (React.createElement("div")), whereas a component you’ve created is not (React.createElement(MyComponent)).
2. The types of the attributes being passed in the JSX element should be looked up differently. Intrinsic element attributes should be known intrinsically whereas components will likely want to specify their own set of attributes.

TypeScript uses the [same convention that React does](http://facebook.github.io/react/docs/jsx-in-depth.html#html-tags-vs.-react-components) for distinguishing between these. An intrinsic element always begins with a lowercase letter, and a value-based element always begins with an uppercase letter.

## Intrinsic elements

Intrinsic elements are looked up on the special interface JSX.IntrinsicElements. By default, if this interface is not specified, then anything goes and intrinsic elements will not be type checked. However, if this interface is present, then the name of the intrinsic element is looked up as a property on the JSX.IntrinsicElements interface. For example:

**declare** namespace JSX {

**interface** IntrinsicElements {

foo: any

}

}

<foo />; // ok

<bar />; // error

In the above example, <foo /> will work fine but <bar /> will result in an error since it has not been specified on JSX.IntrinsicElements.

Note: You can also specify a catch-all string indexer on JSX.IntrinsicElements as follows:

**declare** namespace JSX {

**interface** IntrinsicElements {

[elemName: string]: any;

}

}

## Value-based elements

Value based elements are simply looked up by identifiers that are in scope.

**import** MyComponent from "./myComponent";

<**MyComponent** />; // ok

<SomeOtherComponent />; // error

It is possible to limit the type of a value-based element. However, for this we must introduce two new terms: the element class type and the element instance type.

Given <Expr />, the element class type is the type of Expr. So in the example above, if MyComponent was an ES6 class the class type would be that class. If MyComponent was a factory function, the class type would be that function.

Once the class type is established, the instance type is determined by the union of the return types of the class type’s call signatures and construct signatures. So again, in the case of an ES6 class, the instance type would be the type of an instance of that class, and in the case of a factory function, it would be the type of the value returned from the function.

**class** MyComponent {

render() {}

}

// use a construct signature

**var** myComponent = **new** MyComponent();

// element class type => MyComponent

// element instance type => { render: () => void }

**function** **MyFactoryFunction**() {

**return** {

render: () => {

}

}

}

// use a call signature

**var** myComponent = MyFactoryFunction();

// element class type => FactoryFunction

// element instance type => { render: () => void }

The element instance type is interesting because it must be assignable to JSX.ElementClass or it will result in an error. By default JSX.ElementClass is {}, but it can be augmented to limit the use of JSX to only those types that conform to the proper interface.

**declare** namespace JSX {

**interface** ElementClass {

render: any;

}

}

**class** MyComponent {

render() {}

}

**function** **MyFactoryFunction**() {

**return** { render: () => {} }

}

<MyComponent />; // ok

<MyFactoryFunction />; // ok

**class** NotAValidComponent {}

**function** **NotAValidFactoryFunction**() {

**return** {};

}

<NotAValidComponent />; // error

<NotAValidFactoryFunction />; // error

## Attribute type checking

The first step to type checking attributes is to determine the element attributes type. This is slightly different between intrinsic and value-based elements.

For intrinsic elements, it is the type of the property on JSX.IntrinsicElements

**declare** namespace JSX {

**interface** IntrinsicElements {

foo: { bar?: boolean }

}

}

// element attributes type for 'foo' is '{bar?: boolean}'

<foo bar />;

For value-based elements, it is a bit more complex. It is determined by the type of a property on the element instance type that was previously determined. Which property to use is determined by JSX.ElementAttributesProperty. It should be declared with a single property. The name of that property is then used.

**declare** namespace JSX {

**interface** ElementAttributesProperty {

props; // specify the property name to use

}

}

**class** MyComponent {

// specify the property on the element instance type

props: {

foo?: string;

}

}

// element attributes type for 'MyComponent' is '{foo?: string}'

<MyComponent foo="bar" />

The element attribute type is used to type check the attributes in the JSX. Optional and required properties are supported.

**declare** namespace JSX {

**interface** IntrinsicElements {

foo: { requiredProp: string; optionalProp?: number }

}

}

<foo requiredProp="bar" />; // ok

<foo requiredProp="bar" optionalProp={0} />; // ok

<foo />; // error, requiredProp is missing

<foo requiredProp={0} />; // error, requiredProp should be a string

<foo requiredProp="bar" unknownProp />; // error, unknownProp does not exist

<foo requiredProp="bar" some-unknown-prop />; // ok, because 'some-unknown-prop' is not a valid identifier

Note: If an attribute name is not a valid JS identifier (like a data-\* attribute), it is not considered to be an error if it is not found in the element attributes type.

The spread operator also works:

var props = { requiredProp: "bar" };

<foo {...props} />; // ok

var badProps = {};

<foo {...badProps} />; // error

# The JSX result type

By default the result of a JSX expression is typed as any. You can customize the type by specifying the JSX.Elementinterface. However, it is not possible to retrieve type information about the element, attributes or children of the JSX from this interface. It is a black box.

# Embedding Expressions

JSX allows you to embed expressions between tags by surrounding the expressions with curly braces ({ }).

var a = <div>

{["foo", "bar"].map(i => <span>{i / 2}</span>)}

</div>

The above code will result in an error since you cannot divide a string by a number. The output, when using the preserve option, looks like:

var a = <div>

{["foo", "bar"].map(function (i) { return <span>{i / 2}</span>; })}

</div>

# React integration

To use JSX with React you should use the [React typings](https://github.com/DefinitelyTyped/DefinitelyTyped/tree/master/types/react). These typings define the JSX namespace appropriately for use with React.

/// <reference path="react.d.ts" />

**interface** Props {

foo: string;

}

**class** MyComponent extends React.Component<Props, {}> {

render() {

**return** <**span**>{this.props.foo}</**span**>

}

}

<**MyComponent** foo="bar" />; // ok

<MyComponent foo={0} />; // error

# Decorators

# Introduction

With the introduction of Classes in TypeScript and ES6, there now exist certain scenarios that require additional features to support annotating or modifying classes and class members. Decorators provide a way to add both annotations and a meta-programming syntax for class declarations and members. Decorators are a [stage 2 proposal](https://github.com/tc39/proposal-decorators) for JavaScript and are available as an experimental feature of TypeScript.

NOTE  Decorators are an experimental feature that may change in future releases.

To enable experimental support for decorators, you must enable the experimentalDecorators compiler option either on the command line or in your tsconfig.json:

**Command Line**:

tsc --target ES5 --experimentalDecorators

**tsconfig.json**:

{

"compilerOptions": {

"target": "ES5",

"experimentalDecorators": true

}

}

# Decorators

A Decorator is a special kind of declaration that can be attached to a [class declaration](https://www.typescriptlang.org/docs/handbook/decorators.html#class-decorators), [method](https://www.typescriptlang.org/docs/handbook/decorators.html#method-decorators), [accessor](https://www.typescriptlang.org/docs/handbook/decorators.html#accessor-decorators), [property](https://www.typescriptlang.org/docs/handbook/decorators.html#property-decorators), or [parameter](https://www.typescriptlang.org/docs/handbook/decorators.html#parameter-decorators). Decorators use the form @expression, where expression must evaluate to a function that will be called at runtime with information about the decorated declaration.

For example, given the decorator @sealed we might write the sealed function as follows:

**function** **sealed**(target) {

// do something with 'target' ...

}

NOTE  You can see a more detailed example of a decorator in [Class Decorators](https://www.typescriptlang.org/docs/handbook/decorators.html#class-decorators), below.

## Decorator Factories

If we want to customize how a decorator is applied to a declaration, we can write a decorator factory. A Decorator Factoryis simply a function that returns the expression that will be called by the decorator at runtime.

We can write a decorator factory in the following fashion:

**function** **color**(value: string) { // this is the decorator factory

**return** **function** (target) { // this is the decorator

// do something with 'target' and 'value'...

}

}

NOTE  You can see a more detailed example of a decorator factory in [Method Decorators](https://www.typescriptlang.org/docs/handbook/decorators.html#method-decorators), below.

## Decorator Composition

Multiple decorators can be applied to a declaration, as in the following examples:

* On a single line:
* @f @g x
* On multiple lines:
* @f
* @g
* x

When multiple decorators apply to a single declaration, their evaluation is similar to [function composition in mathematics](http://en.wikipedia.org/wiki/Function_composition). In this model, when composing functions f and g, the resulting composite (f ∘ g)(x) is equivalent to f(g(x)).

As such, the following steps are performed when evaluating multiple decorators on a single declaration in TypeScript:

1. The expressions for each decorator are evaluated top-to-bottom.
2. The results are then called as functions from bottom-to-top.

If we were to use [decorator factories](https://www.typescriptlang.org/docs/handbook/decorators.html#decorator-factories), we can observe this evaluation order with the following example:

**function** **f**() {

console.log("f(): evaluated");

**return** **function** (target, propertyKey: string, descriptor: PropertyDescriptor) {

console.log("f(): called");

}

}

**function** **g**() {

console.log("g(): evaluated");

**return** **function** (target, propertyKey: string, descriptor: PropertyDescriptor) {

console.log("g(): called");

}

}

**class** C {

@f()

@g()

method() {}

}

Which would print this output to the console:

f(): evaluated

g(): evaluated

g(): called

f(): called

## Decorator Evaluation

There is a well defined order to how decorators applied to various declarations inside of a class are applied:

1. Parameter Decorators, followed by Method, Accessor, or Property Decorators are applied for each instance member.
2. Parameter Decorators, followed by Method, Accessor, or Property Decorators are applied for each static member.
3. Parameter Decorators are applied for the constructor.
4. Class Decorators are applied for the class.

## Class Decorators

A Class Decorator is declared just before a class declaration. The class decorator is applied to the constructor of the class and can be used to observe, modify, or replace a class definition. A class decorator cannot be used in a declaration file, or in any other ambient context (such as on a declare class).

The expression for the class decorator will be called as a function at runtime, with the constructor of the decorated class as its only argument.

If the class decorator returns a value, it will replace the class declaration with the provided constructor function.

NOTE  Should you chose to return a new constructor function, you must take care to maintain the original prototype. The logic that applies decorators at runtime will **not** do this for you.

The following is an example of a class decorator (@sealed) applied to the Greeter class:

@sealed

**class** Greeter {

greeting: string;

**constructor**(message: string) {

**this**.greeting = message;

}

greet() {

**return** "Hello, " + **this**.greeting;

}

}

We can define the @sealed decorator using the following function declaration:

**function** **sealed**(**constructor**: Function) {

Object.seal(**constructor**);

Object.seal(**constructor**.prototype);

}

When @sealed is executed, it will seal both the constructor and its prototype.

Next we have an example of how to override the constructor.

**function** **classDecorator**<**T** **extends** {**new**(...args:any[]):{}}>(**constructor**:T) {

**return** **class** extends **constructor** {

newProperty = "new property";

hello = "override";

}

}

@classDecorator

**class** Greeter {

property = "property";

hello: string;

**constructor**(m: string) {

**this**.hello = m;

}

}

console.log(**new** Greeter("world"));

## Method Decorators

A Method Decorator is declared just before a method declaration. The decorator is applied to the Property Descriptor for the method, and can be used to observe, modify, or replace a method definition. A method decorator cannot be used in a declaration file, on an overload, or in any other ambient context (such as in a declare class).

The expression for the method decorator will be called as a function at runtime, with the following three arguments:

1. Either the constructor function of the class for a static member, or the prototype of the class for an instance member.
2. The name of the member.
3. The Property Descriptor for the member.

NOTE  The Property Descriptor will be undefined if your script target is less than ES5.

If the method decorator returns a value, it will be used as the Property Descriptor for the method.

NOTE  The return value is ignored if your script target is less than ES5.

The following is an example of a method decorator (@enumerable) applied to a method on the Greeter class:

**class** Greeter {

greeting: string;

**constructor**(message: string) {

**this**.greeting = message;

}

@enumerable(false)

greet() {

**return** "Hello, " + **this**.greeting;

}

}

We can define the @enumerable decorator using the following function declaration:

**function** **enumerable**(value: boolean) {

**return** **function** (target: any, propertyKey: string, descriptor: PropertyDescriptor) {

descriptor.enumerable = value;

};

}

The @enumerable(false) decorator here is a [decorator factory](https://www.typescriptlang.org/docs/handbook/decorators.html#decorator-factories). When the @enumerable(false) decorator is called, it modifies the enumerable property of the property descriptor.

## Accessor Decorators

An Accessor Decorator is declared just before an accessor declaration. The accessor decorator is applied to the Property Descriptor for the accessor and can be used to observe, modify, or replace an accessor’s definitions. An accessor decorator cannot be used in a declaration file, or in any other ambient context (such as in a declare class).

NOTE  TypeScript disallows decorating both the get and set accessor for a single member. Instead, all decorators for the member must be applied to the first accessor specified in document order. This is because decorators apply to a Property Descriptor, which combines both the get and set accessor, not each declaration separately.

The expression for the accessor decorator will be called as a function at runtime, with the following three arguments:

1. Either the constructor function of the class for a static member, or the prototype of the class for an instance member.
2. The name of the member.
3. The Property Descriptor for the member.

NOTE  The Property Descriptor will be undefined if your script target is less than ES5.

If the accessor decorator returns a value, it will be used as the Property Descriptor for the member.

NOTE  The return value is ignored if your script target is less than ES5.

The following is an example of an accessor decorator (@configurable) applied to a member of the Point class:

**class** Point {

**private** \_x: number;

**private** \_y: number;

**constructor**(x: number, y: number) {

**this**.\_x = x;

**this**.\_y = y;

}

@configurable(false)

**get** x() { **return** **this**.\_x; }

@configurable(false)

**get** y() { **return** **this**.\_y; }

}

We can define the @configurable decorator using the following function declaration:

**function** **configurable**(value: boolean) {

**return** **function** (target: any, propertyKey: string, descriptor: PropertyDescriptor) {

descriptor.configurable = value;

};

}

## Property Decorators

A Property Decorator is declared just before a property declaration. A property decorator cannot be used in a declaration file, or in any other ambient context (such as in a declare class).

The expression for the property decorator will be called as a function at runtime, with the following two arguments:

1. Either the constructor function of the class for a static member, or the prototype of the class for an instance member.
2. The name of the member.

NOTE  A Property Descriptor is not provided as an argument to a property decorator due to how property decorators are initialized in TypeScript. This is because there is currently no mechanism to describe an instance property when defining members of a prototype, and no way to observe or modify the initializer for a property. As such, a property decorator can only be used to observe that a property of a specific name has been declared for a class.

If the property decorator returns a value, it will be used as the Property Descriptor for the member.

NOTE  The return value is ignored if your script target is less than ES5.

We can use this information to record metadata about the property, as in the following example:

**class** Greeter {

@format("Hello, %s")

greeting: string;

**constructor**(message: string) {

**this**.greeting = message;

}

greet() {

**let** formatString = getFormat(**this**, "greeting");

**return** formatString.replace("%s", **this**.greeting);

}

}

We can then define the @format decorator and getFormat functions using the following function declarations:

**import** "reflect-metadata";

**const** formatMetadataKey = Symbol("format");

**function** **format**(formatString: string) {

**return** Reflect.metadata(formatMetadataKey, formatString);

}

**function** **getFormat**(target: any, propertyKey: string) {

**return** Reflect.getMetadata(formatMetadataKey, target, propertyKey);

}

The @format("Hello, %s") decorator here is a [decorator factory](https://www.typescriptlang.org/docs/handbook/decorators.html#decorator-factories). When @format("Hello, %s") is called, it adds a metadata entry for the property using the Reflect.metadata function from the reflect-metadata library. When getFormat is called, it reads the metadata value for the format.

NOTE  This example requires the reflect-metadata library. See [Metadata](https://www.typescriptlang.org/docs/handbook/decorators.html#metadata) for more information about the reflect-metadata library.

## Parameter Decorators

A Parameter Decorator is declared just before a parameter declaration. The parameter decorator is applied to the function for a class constructor or method declaration. A parameter decorator cannot be used in a declaration file, an overload, or in any other ambient context (such as in a declare class).

The expression for the parameter decorator will be called as a function at runtime, with the following three arguments:

1. Either the constructor function of the class for a static member, or the prototype of the class for an instance member.
2. The name of the member.
3. The ordinal index of the parameter in the function’s parameter list.

NOTE  A parameter decorator can only be used to observe that a parameter has been declared on a method.

The return value of the parameter decorator is ignored.

The following is an example of a parameter decorator (@required) applied to parameter of a member of the Greeterclass:

**class** Greeter {

greeting: string;

**constructor**(message: string) {

**this**.greeting = message;

}

@validate

greet(@required name: string) {

**return** "Hello " + name + ", " + **this**.greeting;

}

}

We can then define the @required and @validate decorators using the following function declarations:

**import** "reflect-metadata";

**const** requiredMetadataKey = Symbol("required");

**function** **required**(target: Object, propertyKey: string | symbol, parameterIndex: number) {

**let** existingRequiredParameters: number[] = Reflect.getOwnMetadata(requiredMetadataKey, target, propertyKey) || [];

existingRequiredParameters.push(parameterIndex);

Reflect.defineMetadata(requiredMetadataKey, existingRequiredParameters, target, propertyKey);

}

**function** **validate**(target: any, propertyName: string, descriptor: TypedPropertyDescriptor<Function>) {

**let** method = descriptor.value;

descriptor.value = **function** () {

**let** requiredParameters: number[] = Reflect.getOwnMetadata(requiredMetadataKey, target, propertyName);

**if** (requiredParameters) {

**for** (**let** parameterIndex of requiredParameters) {

**if** (parameterIndex >= arguments.length || arguments[parameterIndex] === undefined) {

**throw** **new** Error("Missing required argument.");

}

}

}

**return** method.apply(**this**, arguments);

}

}

The @required decorator adds a metadata entry that marks the parameter as required. The @validate decorator then wraps the existing greet method in a function that validates the arguments before invoking the original method.

NOTE  This example requires the reflect-metadata library. See [Metadata](https://www.typescriptlang.org/docs/handbook/decorators.html#metadata) for more information about the reflect-metadata library.

## Metadata

Some examples use the reflect-metadata library which adds a polyfill for an [experimental metadata API](https://github.com/rbuckton/ReflectDecorators). This library is not yet part of the ECMAScript (JavaScript) standard. However, once decorators are officially adopted as part of the ECMAScript standard these extensions will be proposed for adoption.

You can install this library via npm:

npm i reflect-metadata --save

TypeScript includes experimental support for emitting certain types of metadata for declarations that have decorators. To enable this experimental support, you must set the emitDecoratorMetadata compiler option either on the command line or in your tsconfig.json:

**Command Line**:

tsc --target ES5 --experimentalDecorators --emitDecoratorMetadata

**tsconfig.json**:

{

"compilerOptions": {

"target": "ES5",

"experimentalDecorators": true,

"emitDecoratorMetadata": true

}

}

When enabled, as long as the reflect-metadata library has been imported, additional design-time type information will be exposed at runtime.

We can see this in action in the following example:

**import** "reflect-metadata";

**class** Point {

x: number;

y: number;

}

**class** Line {

**private** \_p0: Point;

**private** \_p1: Point;

@validate

**set** p0(value: Point) { **this**.\_p0 = value; }

**get** p0() { **return** **this**.\_p0; }

@validate

**set** p1(value: Point) { **this**.\_p1 = value; }

**get** p1() { **return** **this**.\_p1; }

}

**function** **validate**<**T**>(target: any, propertyKey: string, descriptor: TypedPropertyDescriptor<T>) {

**let** **set** = descriptor.set;

descriptor.set = **function** (value: T) {

**let** **type** = Reflect.getMetadata("design:type", target, propertyKey);

**if** (!(value **instanceof** **type**)) {

**throw** **new** TypeError("Invalid type.");

}

**set**(value);

}

}

The TypeScript compiler will inject design-time type information using the @Reflect.metadata decorator. You could consider it the equivalent of the following TypeScript:

**class** Line {

**private** \_p0: Point;

**private** \_p1: Point;

@validate

@Reflect.metadata("design:type", Point)

**set** p0(value: Point) { **this**.\_p0 = value; }

**get** p0() { **return** **this**.\_p0; }

@validate

@Reflect.metadata("design:type", Point)

**set** p1(value: Point) { **this**.\_p1 = value; }

**get** p1() { **return** **this**.\_p1; }

}

NOTE  Decorator metadata is an experimental feature and may introduce breaking changes in future releases.

# Mixins

# Introduction

Along with traditional OO hierarchies, another popular way of building up classes from reusable components is to build them by combining simpler partial classes. You may be familiar with the idea of mixins or traits for languages like Scala, and the pattern has also reached some popularity in the JavaScript community.

# Mixin sample

In the code below, we show how you can model mixins in TypeScript. After the code, we’ll break down how it works.

// Disposable Mixin

**class** Disposable {

isDisposed: boolean;

dispose() {

**this**.isDisposed = true;

}

}

// Activatable Mixin

**class** Activatable {

isActive: boolean;

activate() {

**this**.isActive = true;

}

deactivate() {

**this**.isActive = false;

}

}

**class** SmartObject **implements** Disposable, Activatable {

**constructor**() {

setInterval(() => console.log(**this**.isActive + " : " + **this**.isDisposed), 500);

}

interact() {

**this**.activate();

}

// Disposable

isDisposed: boolean = false;

dispose: () => void;

// Activatable

isActive: boolean = false;

activate: () => void;

deactivate: () => void;

}

applyMixins(SmartObject, [Disposable, Activatable]);

**let** smartObj = **new** SmartObject();

setTimeout(() => smartObj.interact(), 1000);

////////////////////////////////////////

// In your runtime library somewhere

////////////////////////////////////////

**function** **applyMixins**(derivedCtor: any, baseCtors: any[]) {

baseCtors.forEach(baseCtor => {

Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {

derivedCtor.prototype[name] = baseCtor.prototype[name];

});

});

}

# Understanding the sample

The code sample starts with the two classes that will act as our mixins. You can see each one is focused on a particular activity or capability. We’ll later mix these together to form a new class from both capabilities.

// Disposable Mixin

**class** Disposable {

isDisposed: boolean;

dispose() {

**this**.isDisposed = true;

}

}

// Activatable Mixin

**class** Activatable {

isActive: boolean;

activate() {

**this**.isActive = true;

}

deactivate() {

**this**.isActive = false;

}

}

Next, we’ll create the class that will handle the combination of the two mixins. Let’s look at this in more detail to see how it does this:

**class** SmartObject **implements** Disposable, Activatable {

The first thing you may notice in the above is that instead of using extends, we use implements. This treats the classes as interfaces, and only uses the types behind Disposable and Activatable rather than the implementation. This means that we’ll have to provide the implementation in class. Except, that’s exactly what we want to avoid by using mixins.

To satisfy this requirement, we create stand-in properties and their types for the members that will come from our mixins. This satisfies the compiler that these members will be available at runtime. This lets us still get the benefit of the mixins, albeit with some bookkeeping overhead.

// Disposable

isDisposed: boolean = false;

dispose: () => void;

// Activatable

isActive: boolean = false;

activate: () => void;

deactivate: () => void;

Finally, we mix our mixins into the class, creating the full implementation.

applyMixins(SmartObject, [Disposable, Activatable]);

Lastly, we create a helper function that will do the mixing for us. This will run through the properties of each of the mixins and copy them over to the target of the mixins, filling out the stand-in properties with their implementations.

**function** **applyMixins**(derivedCtor: any, baseCtors: any[]) {

baseCtors.forEach(baseCtor => {

Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {

derivedCtor.prototype[name] = baseCtor.prototype[name];

});

});

}

# Triple-Slash Directives

Triple-slash directives are single-line comments containing a single XML tag. The contents of the comment are used as compiler directives.

Triple-slash directives are **only** valid at the top of their containing file. A triple-slash directive can only be preceded by single or multi-line comments, including other triple-slash directives. If they are encountered following a statement or a declaration they are treated as regular single-line comments, and hold no special meaning.

## /// <reference path="..." />

The /// <reference path="..." /> directive is the most common of this group. It serves as a declaration of dependency between files.

Triple-slash references instruct the compiler to include additional files in the compilation process.

They also serve as a method to order the output when using --out or --outFile. Files are emitted to the output file location in the same order as the input after preprocessing pass.

### *Preprocessing input files*

The compiler performs a preprocessing pass on input files to resolve all triple-slash reference directives. During this process, additional files are added to the compilation.

The process starts with a set of root files; these are the file names specified on the command-line or in the "files" list in the tsconfig.json file. These root files are preprocessed in the same order they are specified. Before a file is added to the list, all triple-slash references in it are processed, and their targets included. Triple-slash references are resolved in a depth first manner, in the order they have been seen in the file.

A triple-slash reference path is resolved relative to the containing file, if unrooted.

### *Errors*

It is an error to reference a file that does not exist. It is an error for a file to have a triple-slash reference to itself.

### *Using --noResolve*

If the compiler flag --noResolve is specified, triple-slash references are ignored; they neither result in adding new files, nor change the order of the files provided.

## /// <reference types="..." />

Similar to a /// <reference path="..." /> directive, this directive serves as a declaration of dependency; a /// <reference types="..." />, however, declares a dependency on an @types package.

Including /// <reference types="node" /> in a declaration file declares that this file uses names declared in @types/node/index.d.ts; and thus, this package needs to be included in the compilation along with the declaration file.

The process of resolving these @types package names are similar to process of resolving module names in an importstatement. An easy way to think of triple-slash-reference-types directives as an import for declaration packages.

Use these directives only when you’re authoring a d.ts file by hand.

For declaration files generated during compilation, the compiler will automatically add /// <reference types="..." /> for you; A /// <reference types="..." /> in a generated declaration file is added if and only if the resulting file uses any declarations from the referenced @types package.

For declaring a dependency on an @types package in a .ts file, use --types on the command line or in your tsconfig.json instead. See [using @types, typeRoots and types in tsconfig.json files](https://www.typescriptlang.org/docs/handbook/tsconfig-json.html#types-typeroots-and-types) for more details.

## /// <reference no-default-lib="true"/>

This directive marks a file as a default library. You will see this comment at the top of lib.d.ts and its different variants.

This directive instructs the compiler to not include the default library (i.e. lib.d.ts) in the compilation. The impact here is similar to passing --noLib on the command line.

Also note that when passing --skipDefaultLibCheck, the compiler will only skip checking files with /// <reference no-default-lib="true"/>.

## /// <amd-module />

By default AMD modules are generated anonymous. This can lead to problems when other tools are used to process the resulting modules, such as bundlers (e.g. r.js).

The amd-module directive allows passing an optional module name to the compiler:

##### ***amdModule.ts***

///<amd-module name="NamedModule"/>

**export** **class** C {

}

Will result in assigning the name NamedModule to the module as part of calling the AMD define:

##### ***amdModule.js***

define("NamedModule", ["require", "exports"], **function** (require, exports) {

**var** C = (**function** () {

**function** **C**() {

}

**return** C;

})();

exports.C = C;

});

## /// <amd-dependency />

**Note**: this directive has been deprecated. Use import "moduleName"; statements instead.

/// <amd-dependency path="x" /> informs the compiler about a non-TS module dependency that needs to be injected in the resulting module’s require call.

The amd-dependency directive can also have an optional name property; this allows passing an optional name for an amd-dependency:

/// <amd-dependency path="legacy/moduleA" name="moduleA"/>

**declare** **var** moduleA:MyType

moduleA.callStuff()

Generated JS code:

define(["require", "exports", "legacy/moduleA"], **function** (require, exports, moduleA) {

moduleA.callStuff()

});

## [**Typescript의 Decorator - 1. Class Decorator**](http://partnerjun.tistory.com/61)

2017.08.13 12:52

타입스크립트의 데코레이터는 자바에서의 어노테이션과 유사하게 메타데이터를 지정할 수 있다. 하지만 인터페이스로써 메타데이터만을 저장하는 자바와 달리 데코레이터는 메소드나 프로퍼티를 직접 변경할 수 있다.

하지만 어노테이션이 붙은 모든 클래스나 메소드, 프로퍼티를 찾을 수 있는 자바 리플렉션과 달리 [**'reflect-metadata'**](https://rbuckton.github.io/reflect-metadata/)에는 그런 기능이 없는 것으로 보인다. Angular의 app.module.ts 파일에 컴포넌트 클래스명을 직접 입력해야 하는 이유가 이것 때문이 아닐까 하는 추측을 해본다(추측만).

몇 가지 예제에 사용할 클래스는 아래와 같다.

**class** Car {  
 name: **string**;  
 price: **number**;  
 type: **string**;  
  
 **constructor**(*name*: **string**, *price*: **number**) {  
 **this**.name = *name*;  
 **this**.price = *price*;  
 }  
  
}

**1. 클래스 데코레이터**

클래스 선언부 앞에 붙이고, 클래스의 프로퍼티를 변경하는 데코레이터다.

@ClassDecorator1  
**class** Car {  
 name: **string**;

...

데코레이터도 자바에서의 어노테이션과 마찬가지로 '@데코레이터명'을 특정 위치에 기입한다.

**function** ClassDecorator1(*constructor*: **any**) {  
 console.log(*constructor*); // ƒ Car(name, price) {  
 // this.name = name;  
 // this.price = price;  
 // }  
 **return** <**any**>**class extends** *constructor* {  
 name = 'SM6';  
 color = 'black';  
 }  
}

소스를 보면 알 수 있듯 생성자를 오버라이드하여 새로운 클래스를 반환한다.

이 코드에서는 생성자 파라미터를 any 타입으로 받았는데, [**공식 홈페이지에 따르면**](https://www.typescriptlang.org/docs/handbook/decorators.html) 사실 any타입이 아니라

<T **extends** {**new**(...*args*:**any**[]):{}}>(*constructor*:T)

...args: any[] 파라미터를 받는 생성자를 가진 타입 T로 받아야 한다.

**let** myCar = **new** Car('SM5', 2000);  
console.log(myCar); // class\_1 {name: "SM6", price: 2000, color: "black"}

클래스 선언과 달리 color 프로퍼티가 추가된 것을 확인할 수 있다.

**2. 파라미터를 받는 클래스 데코레이터**

앵귤러의 @Component 데코레이터처럼 데코레이터에 파라미터를 입력받을 수도 있다.

@ClassDecorator2({someValue: 'hello!'})  
**class** Car {  
 name: **string**;

...

파라미터를 입력하는 데코레이터도 자바의 어노테이션과 마찬가지로 ()안에 파라미터를 입력한다.

**function** ClassDecorator2(*param*: **any**) {  
 console.log(*param*); // {someValue: "hello!"}  
 **return function**(*constructor*: **any**) {  
 console.log(*constructor*); // ƒ Car(name, price) {  
 // this.name = name;  
 // this.price = price;  
 // }  
 **return** <**any**>**class extends** *constructor* {  
 someValue = *param*.someValue + ' world!';  
 }  
 }  
}

첫 번째 예제와 비슷하지만 파라미터를 받는 데코레이터 선언부와 생성자를 받는 데코레이터 선언부를 따로 선언한다.

**let** myCar2 = **new** Car('SM5', 2000);  
console.log(myCar2); // class\_2 {name: "SM5", price: 2000, someValue: "hello! world!"}

데코레이터에 파라미터로 입력한 "hello!"와 데코레이터 내부의 프로퍼티 변경으로 Car 클래스의 객체에 {someValue: "hello! world!"} 가 입력된 것을 확인할 수 있다.

|  |
| --- |
| **function** ClassDecorator1(param: **any**) {  console.log(param); // ƒ Car(name, price) {  // this.name = name;  // this.price = price;  // }  **return function**(constructor: **any**) {  **return** <**any**>**class extends** constructor {  name = **this**.name || 'S2M6';  color = **this**.color || 'b2lack';  somValue = param.somValue;  }  } }  @ClassDecorator1({somValue: 'hello'}) **class** Car {  name: **string**;  price: **number**;  type: **string**;   **constructor**(name: **string**, price: **number**) {  **this**.name = name;  **this**.price = price;  }  } console.log(**new** Car('vvv',11))  { somValue: 'hello' }  class\_1 { name: 'vvv', price: 11, color: undefined, somValue: 'hello' } |
|  |

# 07. Typescript 데코레이터(Decorator)

2017.08.12 in [Typescript](https://skout90.github.io/categories/Typescript/)

tsonfig.json에서 "experimentalDecorators": true 명시해줘야 사용가능

## Class Decorator

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 | // 인자가 없을 시엔, 생성자 함수를 선언해줘야 function hello(constructFn: Function) {  console.log(constructFn); }  @hello class Person { } |

## Method Decorator

**read only 데코레이터 예제**

함수를 재작성할 수 없게 함.

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 | function editable(canBeEditable: boolean) {  return function (target: any, propName: string, description: PropertyDescriptor) {  description.writable = canBeEditable;  } }  class Person {  @editable(false)  hello() {  console.log('hello');  } }  const p = new Person(); p.hello(); // hello p.hello = function () {  console.log('world'); } p.hello(); // hello |

target/propName/PropertyDescriptor를 파라미터로 받는 함수를 리턴해주어야 함.

## Property Decorator

target/propName을 파라미터로 받는 함수를 리턴해주어야 함.

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | function writable(canBeWritable: boolean) {  return function (target: any, propName: string): any {  return {  writable: canBeWritable  };  } }  class Person {  @writable(false)  name: string = "junho"; }  const p = new Person(); console.log(p.name); // junho p.name = 'dd'; // error! |

## Parameter Decorator

target/methodName/paramIndex를 파라미터로 가져야함

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 | function printInfo(target: any, methodName: string, paramIndex: number) {  console.log(target);  console.log(methodName);  console.log(paramIndex); }  class Person {  private \_name: string;  private \_age: number;   constructor(name: string, @printInfo age: number) {  this.\_name = name;  this.\_age = age;  }   hello( @printInfo message: string) {  console.log(message);  } } |

*결과*

*Person {}*

*hello*

*0*

*[Funcion: Person]*

*undefined*

*1*

## Reference

<https://www.inflearn.com/course-status-2/>

<http://slides.com/woongjae/deck-8-15#/3/3>

Class Decorator

|  |
| --- |
| **function** hello(constructorFn: Function) {  console.log(constructorFn); }  **function** helloFactory(show: **boolean**) {  **if** (show) {  **return** hello;  } **else** {  **return null**;  } }  // @hello @helloFactory(**true**) **class** Person {  **constructor**() {  console.log('new Person()');  } }  **new** Person();  /\*   helloFactory 는 팩토리 스타일   \*/  먼저 @ 로 선언된 함수쪽에 데이터가 전송되고 (true) 그뒤에 hello의 값처리가된다. "출력: [Function: Person]"  아래쪽에 new Person()이 타기전에 위에내용이 타는것이다.  그뒤에 new Person()을 하게되면 그때 Person의 constructor(생성자)가 실행된다. |

Class Decorator Advanced

|  |
| --- |
| **function** addHello(constructorFn: Function) {  constructorFn.prototype.hello = **function**() {  console.log('hello');  } }  @addHello **class** Person {  **constructor**() {  console.log('new Person()');  } }  **const** person = **new** Person(); (<**any**>person).hello(); |

Method Decorator

|  |
| --- |
| **function** editable(canBeEdit: **boolean**) {   **return function**(target: **any**, propName: **string**, description: PropertyDescriptor) {  console.log(canBeEdit);  console.log(target);  console.log(propName);  console.log(description);  description.writable = canBeEdit;  } }  **class** Person {  **constructor**() {  console.log('new Person()');  }   @editable(**true**)  hello() {  console.log('hello');  } }  **const** person = **new** Person(); person.hello(); person.hello = **function**() {  console.log('world'); } person.hello(); |

Property Decorator

|  |
| --- |
| **function** writable(canBeWrite: **boolean**) {  **return function**(target: **any**, propName: **string**): **any** {  console.log(canBeWrite);  console.log(target);  console.log(propName);  **return** {  writable: canBeWrite  }  } }  **class** Person {  @writable(**false**)  name: **string** = 'Mark';   **constructor**() {  console.log('new Person()');  } }  **const** person = **new** Person(); console.log(person.name);  /\*   undefined   \*/ |

Parameter Decorator

|  |
| --- |
| **function** printInfo(target: **any**, methodName: **string**, paramIndex: **number**) {  console.log(target);  console.log(methodName);  console.log(paramIndex); }  **class** Person {  **private** \_name: **string**;  **private** \_age: **number**;   **constructor**(name: **string**, @printInfo age: **number**) {  **this**.\_name = name;  **this**.\_age = age;  }   hello(@printInfo message: **string**) {  console.log(message);  } }  /\*   Person { hello: [Function] }  hello  0  [Function: Person]  undefined  1   \*/ |

interface 차이점

|  |
| --- |
| **type** PersonAlias = {  name: **string**;  age: **number**; };  **interface** IPerson **extends** PersonAlias {  }  **let** ip: IPerson = {  name: 'Mark',  age: 35 };  **class** PersonImpl **implements** PersonAlias {  name: **string**;  age: **number**;   hello() {  console.log('안녕하세요');  } }  **let** pi: PersonImpl = **new** PersonImpl(); pi.hello();  **class** PersonChild **extends** PersonAlias {  }  /\*   1. 당연한건 type alias 끼리는 extends, implements 불가  2. interface extends type alias 가능  3. class implements type alias 가능  4. class extends type alias 블가 (interface 도 마찬가지)  5. 마치 interface 처럼 동작한다.   \*/ |

interface - optional property (1)

|  |
| --- |
| **interface** Person {  name: **string**;  age?: **number**; }  **function** hello(person: Person): **void** {  console.log(`안녕하세요! ${person.name} 입니다.`); }  **const** p1: Person = {  name: 'Mark',  age: 35 };  **const** p2: Person = {  name: 'Anna' };  hello(p1); // 안녕하세요! Mark 입니다. hello(p2); // 안녕하세요! Anna 입니다. |

interface - optional property (2)

|  |
| --- |
| **interface** Person {  name: **string**;  age?: **number**;  [props: **string**]: **any**; }  **function** hello(person: Person): **void** {  console.log(`안녕하세요! ${person.name} 입니다.`); }  **const** p1: Person = {  name: 'Mark',  age: 35, };  **const** p2: Person = {  name: 'Anna',  systers: [  'Sung',  'Chan'  ] };  **const** p3: Person = {  name: 'Bokdaengi',  father: p1,  mother: p2 };  hello(p1); // 안녕하세요! Mark 입니다. hello(p2); // 안녕하세요! Anna 입니다. hello(p3); // 안녕하세요! Bokdaengi 입니다. |

interface - function in interface

|  |
| --- |
| **interface** Person {  name: **string**;  age: **number**;  hello(): **void**; }  **const** p1: Person = {  name: 'Mark',  age: 35,  hello: **function** (): **void** {  console.log(**this**);  console.log(`안녕하세요! ${**this**.name} 입니다.`);  } };  **const** p2: Person = {  name: 'Mark',  age: 35,  hello(): **void** {  console.log(**this**);  console.log(`안녕하세요! ${**this**.name} 입니다.`);  } };  **const** p3: Person = {  name: 'Mark',  age: 35,  hello: (): **void** => {  console.log(**this**);  console.log(`안녕하세요! ${**this**.name} 입니다.`);  } };  p1.hello(); // 안녕하세요! Mark 입니다. p2.hello(); // 안녕하세요! Mark 입니다. p3.hello(); // 안녕하세요! 입니다. |

class implements interface

|  |
| --- |
| **interface** IPerson {  name: **string**;  age?: **number**;  hello(): **void**; }  **class** Person **implements** IPerson {  name: **string**;   **constructor**(name: **string**) {  **this**.name = name;  }   hello(): **void** {  console.log(`안녕하세요! ${**this**.name} 입니다.`);  } }  **const** person = **new** Person('Mark'); person.hello(); // 안녕하세요! Mark 입니다. |

interface extends interface

|  |
| --- |
| **interface** Person {  name: **string**;  age?: **number**; }  **interface** Korean **extends** Person {  city: **string**; }  **const** k: Korean = {  name: '이웅재',  city: '서울' }; |

function interface

|  |
| --- |
| **interface** HelloPerson {  // (name: string, age: number): void;  (name: **string**, age?: **number**): **void**; }  **let** helloPerson: HelloPerson = **function** (name: **string**) {  console.log(`안녕하세요! ${name} 입니다.`); };  helloPerson('Mark'); // 안녕하세요! Mark 입니다.  /\*   함수의 타입 체크는 할당할때가 아니라 사용할때 한다는 점을 명심   \*/ |

**Indexable Types**

string OR number

|  |
| --- |
| **interface** StringArray {  [index: **number**]: **string**; }  **const** sa: StringArray = {}; // 옵셔널하다 sa[100] = '백';  **interface** StringDictionary {  [index: **string**]: **string**; }  **const** sd: StringDictionary = {}; // 옵셔널하다 sd.hundred = '백';  **interface** StringArrayDictionary {  [index: **number**]: **string**;  [index: **string**]: **string**; }  **const** sad: StringArrayDictionary = {}; // 당연히 옵셔널하다. sad[100] = '백'; sad.hundred = '백'; sad["vvv"] = "ggg"; |

string index = optional property

|  |
| --- |
| **interface** StringDictionary {  [index: **string**]: **string**;  name: **string**; }  **const** sd: StringDictionary = {  name: '이름' // 필수 };  sd.any = 'any'; // 어떤 프로퍼티도 가능  ////////////////////////////////////////////////  **interface** StringDictionaryNo {  [index: **string**]: **string**;  // name: number; // (X) 인덱서블 타입이 string 값을 가지기 때문에 number 를 필수로 끌어오면 에러 } |

클래스 만들기

|  |
| --- |
| **class** Person {  name: **string**;  age: **number**; }  **const** person: Person = **new** Person(); console.log(person); // Person {} person.age = 35; console.log(person.name); // undefined  /\*   1. 생성자 함수가 없으면, 디폴트 생성자가 불린다.   2. 클래스의 프로퍼티 혹은 멤버 변수가 정의되어 있지만, 값을 대입하지 않으면 undefined 이다.  => 오브젝트에 프로퍼티가 아예 존재하지 않는다.   3. 접근제어자 (Access Modifier) 는 public 이 디폴트 이다.   \*/ |

클래스와 프로퍼티 (1)

|  |
| --- |
| **class** Person {  name: **string**;  age: **number**;   **constructor**() {  console.log(**this**.name === **null**); // false  console.log(**this**.name === undefined); // true  } }  **const** person: Person = **new** Person(); person.name = 'Mark'; person.age = 35; console.log(person); // Person {name: 'mark', age: 35} |

클래스와 프로퍼티 (2)

|  |
| --- |
| **class** Person {  name: **string** = 'Mark';  age: **number** = 35;   **constructor**() {  console.log(**this**.name); // 'mark'  } }  **const** person: Person = **new** Person(); console.log(person); // Person {name: 'Mark', age: 35}  /\*   1. 클래스의 프로퍼티를 선언과 동시에 값을 할당하는 방법도 있다.   2. 생성자가 불리기 전에 이미 프로퍼티의 값이 저장되어 있음을 알 수 있다.   \*/ |

클래스와 프로퍼티의 접근 제어자 (1)

|  |
| --- |
| **class** Person {  **public** name: **string**;  **private** \_age: **number**;   **constructor**(age: **number**) {  **this**.\_age = age;  } }  **const** person: Person = **new** Person(35); person.name = 'Mark'; // person.\_age (X) console.log(person); // Person {name: 'Mark', \_age: 35}  /\*   1. private 으로 설정된 프로퍼티는 dot 으로 접근할 수 없다.   2. 클래스 내부에서는 private 프로퍼티를 사용할 수 있다.   3. private 이 붙은 변수나 함수는 \_ 를 이름앞에 붙이는데,  이는 문법이 아니라 널리 쓰이는 코딩 컨벤션이다.   \*/ |

클래스와 프로퍼티의 접근 제어자 (2)

|  |
| --- |
| **class** Parent {  **private** privateProp: **string**;  **protected** protectedProp: **string**;   **constructor**() {   } }  **class** Child **extends** Parent {  **constructor**() {  **super**();   **this**.protectedProp = 'protected';  // this.privateProp = 'private'; // (X)  } }  /\*   1. 부모에서 private 으로 설정된 프로퍼티는 상속을 받은 자식에서도 접근할 수 없다.   2. 부모에서 protected 로 설정된 프로퍼티는 상속을 받은 자식에서 접근이 가능하다.   3. 상속을 받은 자식 클래스에서 부모 클래스에 this 를 통해 접근하려면, 생성자에서 super(); 를 통해 초기화 해야한다.   \*/ |

클래스와 디폴트 생성자

|  |
| --- |
| **class** Person {  **public** name: **string**;  **private** \_age: **number**;   **constructor**(age: **number**) {  **this**.\_age = age;  } }  **const** person: Person = **new** Person();  /\*   1. 디폴트 생성자는 프로그래머가 만든 생성자가 없을 때 사용할 수 있다.  => 사용자가 만든 생성자가 하나라도 있으면, 디폴트 생성자는 사라진다.   \*/ |

클래스와 메서드

|  |
| --- |
| **class** Person {  **constructor**(**private** \_name: **string**, **private** \_age: **number**) { }   print(): **void** {  console.log(`이름은 ${**this**.\_name} 이고, 나이는 ${**this**.\_age} 살 입니다.`);  }   printName = (): **void** => {  console.log(`이름은 ${**this**.\_name} 입니다.`);  }   **private** printAge(): **void** {  console.log(`나이는 ${**this**.\_age} 살 입니다.`);  } }  **const** person: Person = **new** Person('Mark', 35); person.print(); // 이름은 Mark 이고, 나이는 35 살 입니다. person.printName(); // 이름은 Mark 입니다. // person.printAge(); // (X)  /\*   1. 클래스 내부에 작성된 메서드는 public 이 디폴트  2. arrow function 으로 작성 가능  3. private 을 이용하면 클래스 외부애서 접근 불가   \*/ |

클래스와 상속 (1)

|  |
| --- |
| **class** Parent {  **constructor**(**protected** \_name: **string**, **protected** \_age: **number**) { }   print(): **void** {  console.log(`이름은 ${**this**.\_name} 이고, 나이는 ${**this**.\_age} 살 입니다.`);  }   printName = (): **void** => {  console.log(`이름은 ${**this**.\_name} 입니다.`);  }   **private** printAge(): **void** {  console.log(`나이는 ${**this**.\_age} 살 입니다.`);  } }  **class** Child **extends** Parent {  \_name = 'Mark Jr.'; }  // const p: Child = new Child(); // (X) **const** p: Child = **new** Child('vvv', 5); p.print(); // 이름은 Mark Jr. 이고, 나이는 5 살 입니다.  /\*   1. 상속은 extends 키워드를 이용한다.  2. 자식 클래스에서 디폴트 생성자는 부모의 생성자와 입력 형태가 같다.   \*/  Child클래스의 \_name값이 지정되어있으면 아무리 디폴트 생성자에서 셋을하게되어있지만 처리안된다. |

클래스와 상속 (2)

|  |
| --- |
| **class** Parent {  **constructor**(**protected** \_name: **string**, **private** \_age: **number**) { }   print(): **void** {  console.log(`이름은 ${**this**.\_name} 이고, 나이는 ${**this**.\_age} 살 입니다.`);  }   **protected** printName = (): **void** => {  console.log(`이름은 ${**this**.\_name} 입니다.`);  }   **protected** printAge(): **void** {  console.log(`나이는 ${**this**.\_age} 살 입니다.`);  } }  **class** Child **extends** Parent {  **constructor**(age: **number**) {  **super**('Mark Jr.', age);   **this**.printName();  **this**.printAge();  } }  **const** p: Child = **new** Child(1); // 이름은 Son 입니다. // 나이는 1 살 입니다.  /\*   1. 생성자를 정의하고, this 를 사용하려면, super 를 통해 부모의 생성자를 호출해줘야 한다.  2. super 를 호출할때는 부모 생성자의 입력 타입이 같아야 한다.  3. super 를 호출하는 것은 클래스 외부에서 호출하는 것과 같다.  4. protected 함수를 호출해서 그 안의 private 을 출력하는 것에 주의한다.   \*/ |

클래스와 getter, setter

|  |
| --- |
| **class** Person {  **private** \_name: **string**;  **private** \_age: **number**;   **constructor**(name: **string**, age: **number**) {  **this**.\_name = name;  **this**.\_age = age;  }   **get** name() {  **return this**.\_name;  }   **set** name(name: **string**) {  // 작업  **this**.\_name = `${name} Lee`;  } }  **const** person: Person = **new** Person('Mark', 35);  console.log(person.name); person.name = 'Woongjae'; console.log(person.name);  /\*   1. \_ 를 변수명 앞에 붙이고, 내부에서만 사용한다.  2. getter 를 함수처럼 설정하면, 프로퍼티처럼 꺼내쓸수있다.  3. 마찬가지로 setter 를 함수처럼 설정하면, 추가 작업을 하고 셋팅할 수 있다.   \*/ |

클래스와 static 프로퍼티 => 클래스 멤버 변수

|  |
| --- |
| **class** Person {  **public static** *CITY* = "";  **private static** *lastName*: **string** = 'Lee';  **private** \_name: **string**;  **private** \_age: **number**;   **constructor**(name: **string**, age: **number**) {  **this**.\_name = name;  **this**.\_age = age;  }   **public** print() {  console.log(`${**this**.\_name} ${Person.lastName} in ${Person.*CITY*}.`);  } }  **const** person: Person = **new** Person('Mark', 35); Person.*CITY* = 'Seoul'; person.print(); // Mark Lee in Seoul.  /\*   1. static 키워드를 붙힌 프로퍼티는 클래스.프로퍼티로 사용한다.  2. static 프로퍼티에 private, protected 를 붙히면 똑같이 동작한다.   \*/ |
| **var** Person = (**function** () {  **function** Person(name, age) {  **this**.\_name = name;  **this**.\_age = age;  }  Person.prototype.print = **function** () {  console.log(**this**.\_name + " " + Person.lastName + " in " + Person.CITY + ".");  };  **return** Person; }()); Person.CITY = ""; Person.lastName = 'Lee'; **var** person = **new** Person('Mark', 35); Person.CITY = 'Seoul'; person.print(); // Mark Lee in Seoul. /\*   1. static 키워드를 붙힌 프로퍼티는 클래스.프로퍼티로 사용한다.  2. static 프로퍼티에 private, protected 를 붙히면 똑같이 동작한다.   \*/  //# sourceMappingURL=test.js.map |

클래스와 static 메서드 => 클래스 멤버 함수

|  |
| --- |
| **class** Person {  **public static** *Talk*(): **void** {  console.log('안녕하세요.');  } }  Person.*Talk*(); // 안녕하세요. |
| **var** Person = (**function** () {  **function** Person() {  }  Person.Talk = **function** () {  console.log('안녕하세요.');  };  **return** Person; }()); Person.Talk(); // 안녕하세요. //# sourceMappingURL=test.js.map |

모듈에서 private static 프로퍼티 혹은 메서드

|  |
| --- |
| **class** Person {  **private static** *PROPERTY* = '프라이빗 프로퍼티';  **private static** *METHOD*() {  console.log('프라이빗 메서드');  }   **constructor**() {  console.log(Person.*PROPERTY*);  Person.*METHOD*();  } }  //////////////////////////////////////////////  **const** PROPERTY = '모듈 내 변수'; **function** METHOD() {  console.log('모듈 내 함수'); }  **export class** Person {  **constructor**() {  console.log(PROPERTY);  METHOD();  } } |

Abstract Class

|  |
| --- |
| **abstract class** APerson {  **protected** \_name: **string** = 'Mark';  **abstract** setName(name: **string**): **void**; }  **class** Person **extends** APerson {  setName(name: **string**): **void** {  **this**.\_name = name;  } }  // const person = new APerson(); // (X) **const** person = **new** Person();  /\*   1. abstract 키워드가 사용된 클래스는 new 로 생성할 수 없다.  2. abstract 키워드가 사용된 클래스를 상속하면 abstract 키워드가 붙은 함수를 구현해야 한다.   \*/ |

Reflection class Scan

|  |
| --- |
| **module** myContainer {   @magical  **export class** foo {  }   **export class** bar {  }   @magical  **export class** bas {  } }   **function** magical(item: **any**) {  item.isMagical = "indeed"; }  **function** findMagicalInScope(theScope: **any**) {  **for**(**let** prop **in** theScope) {  **if** (theScope[prop]["isMagical"]) {  console.log(`Is ${prop} magical? ${theScope[prop]["isMagical"]}!`);  } **else** {  console.log(`${prop} is not magical. :-(`);  }  } }  findMagicalInScope(myContainer);  findMagicalInScope(myContainer);  Is foo magical? indeed!  bar is not magical. :-(  Is bas magical? indeed!  https://www.laurivan.com/scan-decorated-classes-in-typescript/ |

Class 와 private constructor

|  |
| --- |
| **class** Preference {  **private constructor**() {   } }  // const p: Preference = new Preference(); (X)  /\*   1. 생성자 함수 앞에 접근제어자인 private 을 붙일 수 있다.  2. 외부에서 생성이 불가능하다.   \*/ |

Class 와 싱글톤 패턴

|  |
| --- |
| **class** Preference {  **public static** *getInstance*() {  **if** (Preference.*instance* === **null**) {  Preference.*instance* = **new** Preference();  }   **return** Preference.*instance*;  }  **private static** *instance*: Preference = **null**;  **private constructor**() {   } }  **const** p: Preference = Preference.*getInstance*();  /\*   1. private 생성자를 이용해서 내부에서만 인스턴스 생성이 가능하도록 함.  2. pubilc static 메서드를 통해 private static 인스턴스 레퍼런스를 획득한다.  3. Lazy Loading (Initialization) : 최초 실행시가 아니라, 사용시에 할당을 함   \*/ |
| **var** Preference = (**function** () {  **function** Preference() {  }  Preference.getInstance = **function** () {  **if** (Preference.instance === **null**) {  Preference.instance = **new** Preference();  }  **return** Preference.instance;  };  **return** Preference; }()); Preference.instance = **null**; **var** p = Preference.getInstance(); |

Class 와 readonly

|  |
| --- |
| **class** Person {  **private readonly** \_name: **string** = **null**;  **public readonly** age: **number** = 35;   **constructor**(name: **string**) {  **this**.\_name = name;  }   **public** setName(name: **string**) {  // this.\_name = name; (X)  } }  **const** p: Person = **new** Person('Mark'); console.log(p.age); // p.age = 36; // (X)  /\*   1. private readonly 로 선언된 경우, 생성자에서는 할당이 가능하다.  2. private readonly 로 선언된 경우, 생성자 이외에서는 할당이 불가능하다.  3. public readonly 로 선언된 경우, 클래스 외부에서는 다른값을 할당할 수 없다.  4. 마치 getter 만 있는 경우와 같다.   \*/ |

Exercise (1)

|  |
| --- |
| **function** Car(name) {  **this**.name = name;  **this**.speed = 0;   **this**.honk = **function**() {  console.log("부우우웅");  };   **this**.accelerate = **function**(speed) {  **this**.speed = **this**.speed + speed;  } }  **var** car = **new** Car("BENZ"); car.honk(); console.log(car.speed); car.accelerate(10); console.log(car.speed); |

Exercise (2)

|  |
| --- |
| **var** baseObject = {  width: 0,  length: 0 };  **var** rectangle = Object.create(baseObject); rectangle.width = 8; rectangle.length = 6; rectangle.area = **function**() {  **return this**.width \* **this**.length; };  console.log(rectangle.area()); |

Exercise (3)

|  |
| --- |
| **var** person = {  \_firstName: "" };  // https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Global\_Objects/Object/defineProperty // ES5 이상 Object.defineProperty(person, "firstName", {  get: **function** () {  **return this**.\_firstName;  },  set: **function** (value) {  **if** (value.length > 3) {  **this**.\_firstName = value;  }  **else** {  **this**.\_firstName = "";  }  },  enumerable: **true**,  configurable: **true** });  console.log(person.firstName); person.firstName = "Ma"; console.log(person.firstName); person.firstName = "Maximilian"; console.log(person.firstName); |

**Generic**

any => generic

|  |
| --- |
| **function** helloString(message: **string**): **string** {  **return** message; }  **function** helloNumber(message: **number**): **number** {  **return** message; }  // 더 많은 반복된 함수들 ...  **function** hello(message: **any**): **any** {  **return** message; }  **function** helloGeneric<T>(message: T): T {  **return** message; }  console.log(hello('Mark').length); console.log(hello(35).length);  console.log(helloGeneric(35).toString()); // console.log(helloGeneric<number>(35).toString());  // hello 의 리턴이 any 이기 때문에 타입 헬퍼가 제대로 되지 않음 // helloGeneric 을 사용하면 정상적으로 사용가능 |

basic generic

|  |
| --- |
| **function** helloGeneric<T>(message: T): T {  **return** message; }  **function** hello<T>(message: T): T {  **return** message; }  console.log(hello<**string**>('Hello')); **let** age = hello(35); hello<**number**>('35'); // <-- 컴파일 에러  /\*   1. Generic 타입을 쓰지 않으면, T 로 추론  2. Generic 타입을 쓰면, T 를 확인   \*/ |

Generic Array

|  |
| --- |
| **function** hello<T>(messages: T[]): T {  **return** messages[0]; }  console.log(hello<**string**>(['Hello', 'World']));  /\*   hello 함수의 제네릭 타입을 [] 를 이용하여 배열로 사용할 수 있음   \*/ |

Generic Types

|  |
| --- |
| **type** HelloGeneric = <T>(message: T) => T;  **const** hello: HelloGeneric = <T>(message: T): T => {  **return** message; }  console.log(hello<**string**>('Hello').length);  /\*   구현체에 return T 를 설정하지 않아도,  return false 시 오류가 나지 않는다?   \*/ |

Generic Class

|  |
| --- |
| **class** Person<T> {  **private** \_name: T;  **private** \_age: **number**;   **constructor**(name: T) {  **this**.\_name = name;  } }  **new** Person('Mark'); // new Person<string>(35);  /\*   명시적으로 제네릭 타입을 설정하면 오류   \*/ |

Generic with extends

|  |
| --- |
| **class** Person<T **extends string** | **number**> {  **private** \_name: T;  **private** \_age: T;   **constructor**(name: T) {  **this**.\_name = name;  } }  **new** Person('Mark'); **new** Person(35); // new Person(true);  /\*   T 가 string 또는 number 를 상속받기 때문에 boolean 은 안된다.   \*/ |

Generic with multiple types

|  |
| --- |
| **class** Person<T, K> {  **private** \_name: T;  **private** \_age: K;   **constructor**(name: T, age: K) {  **this**.\_name = name;  **this**.\_age = age;  } }  **new** Person('Mark', 35); |

type lookup system

|  |
| --- |
| **interface** Person {  name: **string**;  age: **number**; }  **const** person: Person = {  name: 'Mark',  age: 35 };  **function** getProperty<T, K **extends keyof** T>(obj: T, key: K): T[K] {  **return** obj[key]; }  **function** setProperty<T, K **extends keyof** T>(obj: T, key: K, value: T[K]): **void** {  obj[key] = value; }  console.log(getProperty(person, 'name')); // console.log(getProperty(person, fullname)); setProperty(person, 'name', 'Anna'); console.log(getProperty(person, 'name')); // setProperty(person, 'name', 24) |

**iterator**

for..of

|  |
| --- |
| es3  - for (var i = 0; i < array.length; i++)  es5  - array.forEach  - return 으로 순회를 탈출할 수 없다.  es6  - for (const item of array)  - 배열에서만 사용이 가능 |

for..in

|  |
| --- |
| 배열을 순회할 때는 사용하지 않을 것  index 가 number 가 아니라 string 으로 나온다.  배열의 프로퍼티를 순회할 수도 있다.  prototype 체인의 프로퍼티를 순회할 수도 있다.  루프가 무작위로 순회할 수도 있다.  for..of 를 쓸 것  객체를 순회할 때  for (const prop of Object.keys(obj)) 도 사용할 수 있다. |

Example

|  |
| --- |
| **const** array = ['first', 'second']; **const** obj = {  name: 'Mark',  age: 35 };  // 배열에 for..of 이용 **for** (**const** item **of** array) {  console.log(**typeof** item + ', ' + item); }  // 배열에 for..in 이용 // item 이 string 타입의 숫자 **for** (**const** item **in** array) {  console.log(**typeof** item + ', ' + item); }   // 객체에 for..of 이용 => 오류 /\*  for (const item of obj) {  console.log(typeof item + ', ' + item);  }  \*/  // 객체에 for..in 이용 **for** (**const** item **in** obj) {  console.log(**typeof** item + ', ' + item); }  // 객체의 keys 들에 for..of 이용 **for** (**const** item **of** Object.keys(obj)) {  console.log(**typeof** item + ', ' + item); } |

target es3 forEach

|  |
| --- |
| **const** array = ['first', 'second'];  // ts array.forEach((item) => {  console.log(item); });  // js array.forEach(**function** (item) {  console.log(item); });  /\*   target 이 es3 인데도 forEach 는 트랜스파일이 되지 않았음.  https://github.com/Microsoft/TypeScript/issues/2410   \*/ |

Symbol.iterator

|  |
| --- |
| * 프로퍼티이며, 함수가 구현되어있으면, iterable 이라고 한다. * Array, Map, Set, String, Int32Array, Uint32Array, etc. 에는 내장된 구현체가 있으므로 이터러블 하다. * 그냥 객체는 이터러블하지 않다. * 이터레이터를 통해 이터러블한 객체의 Symbol.iterator 함수를 호출한다.   https://s3.amazonaws.com/media-p.slid.es/uploads/640576/images/3840031/iteration----consumers_sources.jpg |

Symbol.iterator

|  |
| --- |
| * target : es3 or es5   + Array 에만 for..of 사용 가능   + 일반 객체에 사용하면 오류 * target : es6   + Symbol.iterator 함수를 구현하면 어떤 객체에도 for..of 사용 가능   https://s3.amazonaws.com/media-p.slid.es/uploads/640576/images/3840413/iteration----iteration_protocol.jpg |

lib.es6.d.ts

|  |
| --- |
| **interface** IteratorResult<T> {  done: **boolean**;  value: T; }  **interface** Iterator<T> {  next(value?: **any**): IteratorResult<T>;  **return**?(value?: **any**): IteratorResult<T>;  **throw**?(e?: **any**): IteratorResult<T>; }  **interface** Iterable<T> {  [Symbol.iterator](): Iterator<T>; }  **interface** IterableIterator<T> **extends** Iterator<T> {  [Symbol.iterator](): IterableIterator<T>; } |

Custom Iterable

|  |
| --- |
| **class** CustomIterable **implements** Iterable<**string**> {  **private** \_array: Array<**string**> = ['first', 'second'];   [Symbol.iterator]() {  **var** nextIndex = 0;   **return** {  next: () => {  **return** {  value: **this**.\_array[nextIndex++],  done: nextIndex > **this**.\_array.length  }  }  }  } }  **const** cIterable = **new** CustomIterable();  **for** (**const** item **of** cIterable) {  console.log(item); } |

**decorator**

Decorator 종류

Class Decorator

Method Decorator

Property Decorator

Parameter Decorator

Class Decorator Basic

|  |
| --- |
| **function** hello(constructorFn: Function) {  console.log(constructorFn); }  **function** helloFactory(show: **boolean**) {  **if** (show) {  **return** hello;  } **else** {  **return null**;  } }  // @hello @helloFactory(**true**) **class** Person {  **constructor**() {  console.log('new Person()');  } }  **new** Person();  /\*   helloFactory 는 팩토리 스타일   \*/ |

Class Decorator Advanced

|  |
| --- |
| **function** addHello(constructorFn: Function) {  constructorFn.prototype.hello = **function**() {  console.log('hello');  } }  @addHello **class** Person {  **constructor**() {  console.log('new Person()');  } }  **const** person = **new** Person(); (<**any**>person).hello(); |

Method Decorator

|  |
| --- |
| **function** editable(canBeEdit: **boolean**) {   **return function**(target: **any**, propName: **string**, description: PropertyDescriptor) {  console.log(canBeEdit);  console.log(target);  console.log(propName);  console.log(description);  description.writable = canBeEdit;  } }  **class** Person {  **constructor**() {  console.log('new Person()');  }   @editable(**true**)  hello() {  console.log('hello');  } }  **const** person = **new** Person(); person.hello(); person.hello = **function**() {  console.log('world'); } person.hello(); |
| **function** editable(canBeEdit: **boolean**) {   **return function**(target: **any**, propName: **string**, description: PropertyDescriptor) {  console.log(canBeEdit);  console.log(target);  console.log(propName);  console.log(description);  target[propName] = **function**(){  console.log("babo");  }  //description.writable = canBeEdit;  } }  **class** Person {  **constructor**() {  console.log('new Person()');  }   @editable(**true**)  hello() {  console.log('hello');  } } // **const** person = **new** Person(); person.hello(); // person.hello = function() { // console.log('world'); // } // person.hello(); |

Property Decorator

|  |
| --- |
| **function** writable(canBeWrite: **boolean**) {  **return function**(target: **any**, propName: **string**): **any** {  console.log(canBeWrite);  console.log(target);  console.log(propName);  **return** {  writable: canBeWrite  }  } }  **class** Person {  @writable(**false**)  name: **string** = 'Mark';   **constructor**() {  console.log('new Person()');  } }  **const** person = **new** Person(); console.log(person.name);  /\*   undefined   \*/ |

Parameter Decorator

|  |
| --- |
| **function** printInfo(target: **any**, methodName: **string**, paramIndex: **number**) {  console.log(target);  console.log(methodName);  console.log(paramIndex); }  **class** Person {  **private** \_name: **string**;  **private** \_age: **number**;   **constructor**(name: **string**, @printInfo age: **number**) {  **this**.\_name = name;  **this**.\_age = age;  }   hello(@printInfo message: **string**) {  console.log(message);  } }  /\*   Person { hello: [Function] }  hello  0  [Function: Person]  undefined  1   \*/ |

**Type Inference**

|  |
| --- |
| **타입 추론**  * 기본적으로 타입을 명시적으로 쓰지 않을 때 추론하는 방법에 대한 규칙   + 명시적으로 쓰는 것은 타입 추론이 아니라 코드를 읽기 좋게 하는 지름길 * let 은 기본적으로 우리가 아는 기본 자료형으로 추론 * const 는 리터럴 타입으로 추론   + 오브젝트 타입을 타입을 쓰지 않으면, 프로퍼티는 let 처럼 추론     - const person = {name: 'Mark', age: 35}; 면     - person => {name: string; age: number;} 로 추론 * 대부분은 추론이 쉽다.   + 단순 변수   + structuring, destructuring * array, 함수의 리턴에서는 원하는데로 얻기가 힘들다. |
| **type** StoreType = {  state: InternalStateType,  restoreInputValues: () => **void**,  disposeOldHosts: () => **void** }; |

배열 타입 추론

|  |
| --- |
| **const** array1 = []; **const** array2 = ['a', 'b', 'c']; **const** array3 = ['a', 1, **false**];  **class** Animal {  name: **string**; }  **class** Dog **extends** Animal {  dog: **string**; }  **class** Cat **extends** Animal {  cat: **string**; }  **const** array4 = [**new** Dog(), **new** Cat()]; |

리턴 타입 추론

|  |
| --- |
| **function** hello(message: **string** | **number**) {  **if** (message === 'world') {  **return** 'world';  } **else** {  **return** 0;  } } |

유니온 타입과 타입 가드

|  |
| --- |
| **interface** Person {  name: **string**;  age: **number**; }  **interface** Car {  brand: **string**;  wheel: **number**; }  **function** isPerson(arg: **any**): arg is Person {  **return** arg.name !== undefined; }  **function** hello(arg: Person | Car) {  **if** (isPerson(arg)) {  console.log(arg.name);  // console.log(arg.brand);  } **else** {  // console.log(arg.name);  console.log(arg.brand);  } } |

**React with TypeScript**

**https://github.com/saystone/tic-tac-toe**