**TypeScript Documentation**

1. Basic Types

(1) Tuple

-> the type of a fixed number of elements is known

-> ex. let x: [string, number]

x = ["hello", 10];

-> When accessing an element with a known index, the correct type is retrived, but when accessing an an element outside the set of known indices, a union type is used

-> ex. x[3] = "world" // OK, 'string' can be assigned to 'string | number'

x[4] = true // Error, 'boolean' isn't string | number

(2) Enum

-> a way of giving more friendly names to sets of numeric values

-> can go from a numeric value to the name of that value in the enum

-> ex. enum Color { Red = 1, Green, Blue }

let colorName: string = Color[2]

alert(colorName) // Green

(3) void

-> Declaring variables of type void is not useful because you can only assign undefined or null to them

2. Variable Declaration

(1) Array Destructuring

-> ex. [first, second] = [second, first] // swap variables

let { a, ...b } = o (object destructuring)

let [ a, ...b ] = o (array destructuring)

(2) Property renaming

-> ex. let o = {a: "foo", b: 12, c: "bar"}

let { a: newName1, b: newName2 } = o

-> Above example is the same as

let newName1 = o.a

let newName2 = o.b

(3) Default values

-> Default values let you specify a default value in case a property is undefined

-> ex. function foo(wholeObject: { a: string, b?: number }) {

let { a, b = 1001 } = wholeObject

}

-> function f({ a, b } = { a: "", b: 0 }): void {

f(); // OK, default to { a: "", b: 0 }

}

-> function f({ a, b = 0 } = { a: "" }): void {

f({ a: "yes" }); // OK, default b = 0

f(); // OK, default to { a: "" }, which then defaults b = 0

f({}); // error, 'a' is required if you supply an argument

(4) Spread

-> You can also spread objects

-> Like array spreading, object spreading proceeds from left-to-right. This means that properties that come later in the spread object overwrite properties that come eariler

-> Object spreading only includes own, enumerable properties. The methods cannot be spread

3. Interface

-> ex. function printLabel(labelledObj: { label: string }) {

console.log(labelledObj.label);

}

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

printLabel(myObj)

-> Above example is the same as

-> ex. interface LabelledValue {

label: string;

}

function printLabel(labelledObj: LabelledValue) {

console.log(labelledObj.label);

}

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

printLabel(myObj)

-> the compiler only checks that at least the ones required are present and match the types required

-> type-checker does not require that these properties come in any sort of order, only that properties the interface requires are present and have the required type

(1) Readonly properties

-> Some properties should only be modifiable when an object is first created

-> ex. interface Point {

readonly x: number;

readonly y: number;

}

-> ex. let a: number[] = [1, 2, 4, 5];

let ro: ReadOnlyArray<number> = a;

ro[0] = 12; // error!

ro.push(5); // error!

ro.length = 100; // error!

a = ro; // error!

a = ro as number[]; // OK!

-> The easiest way to remember whether to use readonly or const is to ask whether you're using it on a variable or a property. Variables use const where as properties use readonly

(2) Excess Property Checks

-> Object literals get special treatment and undergo excess property checking when assigning them to other variables, or passing them as arguments

-> If an object literal has any properties that the "target type" doesn't have, you'll get an error

-> ex. let mySquare = creaetSquare({ colour: "red", width: 100 }) // error!

-> To get around these checks, use a type assertion: let mySquare = createSquare({ width: 100, opacity: 0.5 } as SquareConfig);

-> ex. interface SquareConfig {

color?: string;

width?: number;

[propName: string]: any;

}

(3) Function Types

-> Interfaces are also capable of describing function types consisting of only the parameter(both name and type) list and return type

-> ex. interface SearchFunc {

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

}

-> the names of the parameters do not need to match

-> ex. let mySearch: SearchFunc;

mySearch = function(src, sub) { // type is not specified because mySearch's interface is SearchFunc and type can be checked there

let result = src.search(sub);

return result > -1;

} is also possible because function variable is assigned to 'mySearch' whose type(interface) is SearchFunc

(4) Indexable Types

-> Indexable types have an index signature that describes the types we can use to index into the object along with the corresponding return types when indexing

-> ex. interface StringArray {

[index: number]: string;

}

let myArray: StringArray;

myArray = ["Bob", "Fred"];

let myStr: string = myArray[0];

-> There are two types of supported index signatures: string and number. It is possible to support both types of indexers but the type returned from a numeric indexer must be a subtype of the type returned from the string indexer. This is because when indexing with a number, JavaScript will actually convert that to a string before indexing into an object.

-> ex. interface foo {

[numeric\_indexer: number]: boolean;

[string\_indexer: string]: string;

} // Error!

-> While string index signatures are a powerful way to describe the 'dictionary' pattern, they also enforce that all properties match their return type. This is because a string index declares that 'obj.property' is also avaliable as 'obj["property"]'

-> ex. interface NumberDictionary {

[index: string]: number;

length: number,

name: string // When called with obj["name"], it collides with the definition of '[index: string]: number'

}

(5) Implementing an interface

-> Interfaces describe the public side of the class, rather than both the public and private side. This prohibits you from using them to check that a class also has particular types for the private side of the class instance

-> When working with classes and interfaces, it helps to keep in mind that a class has two types: the type of the static side and the type of the instance side

-> ex. interface ClockConstructor {

new (hour: number, minute: number);

}

class Clock implements ClockConstructor {

currentTime: Date;

constructor(h: number, m: number){}

}

-> Above example invokes an error because when a class implements an interface, only the instance side of the class is checked. Since the constructor sits in the static side, it is not included in this check. Instead, you would need to work with the static side of the class directly. In this example, we define two interfaces, ClockConstructor for the constructor and ClockInterface for the instance methods

-> ex. interface ClockConstructor {

new (hour: number, minute: number): ClockInterface;

}

interface ClockInterface {

tick();

}

function createClock(ctor: ClockConstructor, hour: number, minute: number): ClockInterface {

return new ctor(hour, minute);

}

class DigitalClock implements ClockInterface {

constructor(h: number, m: number) {}

tick() {}

}

class AnalogClock implements ClockInterface {

constructor(h: number, m: number) {}

tick() {}

}

let digital = createClock(DigitalClock, 12, 18);

let analog = createClock(AnalogClock, 7, 12);

// CORRECT EXAMPLE: Because createClock's first parameter is of type ClockConstructor, in createClock(AnalogClock, 7, 12), it checks that AnalogClock has the correct constructor signature

(6) Hybrid Types

-> One such example is an object that acts as both a function and an object, with additional properties

-> ex. interface Counter {

(start: number): string;

interval: number;

reset(): void;

}

function getCounter(): Counter {

let counter = <Counter>function (start: number) {};

counter.interval = 12;

counter.reset = function() {};

return counter;

}

let c = getCounter();

c(10);

c.reset();

c.interval = 5.0;

(7) Interfaces extending classes

-> When an interface type extends a class type it inherits the members of the class but not their implementations. It is as if the interface had declared all of the members of the class without providing an implementation. Interfaces inherit even the private and protected members of a base class. This means that when you create an interface that extends a class with private or protected members, that interface type can only be implemented by that class or a subclass of it

-> ex. class Control { private state: any; }

interface SelectableControl extends Control { select(): void; }

class Button extends Control { select() {} }

class TextBox extends Control { select() {} }

class Image { select() {} }

class Location { select() {} }

-> In the above example, SelectableControl contains all of the members of Control, including the private state property. Since state is a private member it is only possible for descendants of Control to implement SelectableControl. This is because only descendants of Control will have a state private member that originates in the same declaration, which is a requirement for private members to be compatible. Within the Control class it is possible to access the state private member through an instance of SelectableControl. Effectively, a SelectableControl acts like a Control that is known to have a select method. The Button and TextBox classes are subtypes of SelectableControl because they both inherit from Control and have a select method but the Image and Location classes are not.

4. Class

(1) Public, private, protected modifiers

-> TypeScipt is a structural type system. When we compare two different types, regardless of where they came from, if the types of all members are compatible, then we say the types themselves are compatible. However, when comparing types that have private and protected members, we treat these types differently. For two types to be considered compatible, if one of them has a private member, then the other must have a private member that originated in the same declaration. The same applies to protected members

-> ex. class Animal {

private name: string;

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

}

class Rhino extends Animal {

constructor() { super("Rhino") }

}

class Employee {

private name: string;

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

}

let animal = new Animal("Goat");

let rhino = new Rhino();

let employee = new Employee("Bob");

animal = rhino; // OK

animal = employee; // Error: 'Animal' and 'Employee' are not compatible because Employee's name and Animal's name are originated from the same declaration

(2) Readonly members

-> You can make properties readonly by using the readonly keyword. Readonly properties must be initialized at their declaration or in the constructor

(3) Parameter Property

-> We can declare and assign the class member variable simultaneously in constructor parameter. The parameter properties should be prefixed with an accessibility modifier or readonly, or both

-> ex. class Octopus {

constructor(public name: string) {}

} // Now, Octopus instances have 'name' public variable. Prefix 'public' is mandatory

(4) Accessors

-> Accessors with a get and no set are automatically inferred to be readonly

(5) Abstract Classes

-> Abstract classes are base classes from which other classes may be derived. They may not be instantiated directly. Unlike an interface, an abstract class may contain implementation details for its members

-> Methods within an abstract class that are marked as abstract do not contain an implementation and must be implemented in derived classes

(6) Constructor functions

-> When you declare a class in TypeScript, you are actually creating multiple declarations at the same time. The first is the type of the instance of the class

-> ex. class Greeter {

greeting: string;

constructor(message: string) {

this.greeting = message;

}

greet() {}

}

let greeter: Greeter;

greeter = new Greeter('World');

-> Here, when we say 'let greeter: Greeter', we're using Greeter as the type of instances of the class Greeter

-> We're also creating another value that we call the constructor function. This is the function that is called when we 'new' up instances of the class. It also contains all of the static members of the class

-> ex. class Greeter {

static standardGreeting = "Hello, there";

greeting: string;

greet() {}

}

let greeter1: Greeter;

greeter1 = new Greeter();

let greeterMaker: typeof Greeter = Greeter;

greeterMaker.standardGreeting = "Hey there!";

let greeter2 = new greeterMaker();

-> The variable 'greeterMaker' will hold the class itself, or said another way its constructor function. Here we use typeof Greeter, that is "give me the type of the Greeter class itself" rather than the instance type. Or, more precisely, "give me the type of the symbol called Greeter", which is the type of the constructor function. This type will contain all of the static members of Greeter along with the constructor that creates instances of the Greeter class

(7) Using a class as an interface

-> As we said in the previous section, a class declaration creates two things: a type representing instances of the class and a constructor function. Because classes create types, you can use them in the same places you would be able to use interfaces

-> ex. class Point {

x: number;

y: number;

}

interface Point3d extends Point {

z: number;

}

let point3d: Point3d = { x: 1, y: 2, z: 4 };

5. Function

(1) Writing the full function type

-> ex. let myAdd: (x: number, y: number) => number = function(x: number, y: number): number { return x + y };

-> A function's type has the same two parts: the type of the arguments and the return type. When writing out the whole function type, both parts are required. The parameter's name is just to help with readability

-> ex. let myAdd: (baseValue: number, increment: number) => number = function(x: number, y: number): number { return x + y; };

-> The following two examples are both correct

-> ex. let myAdd = function(x: number, y: number): number { return x + y; };

-> ex. let myAdd: (baseValue: number, increment: number) => number = function(x, y) { return x + y; };

(2) Optional, Default Parameter

-> Any optional parameters must follow required parameters

-> Default parameters that come after all required parameters are treated as optional and just like optional parameters, can be omitted when calling their respective function

-> ex. function buildName(firstName: string, lastName?:string) {}

-> ex. function buildName(firstName: string, lastName = "smith") {}

-> Above 2 examples share the same type (firstName: string, lastName?: string) => string. The default value of lastName disappears in the type, only leaving behind the fact that the parameter is optional

-> Unlike plain optional parameters, default-initialized parameters don't need to occur after required parameters. If a default-initialized parameter comes before a required parameter, users need to explicitly pass undefined to get the default initialized value

(3) Rest Parameters

-> multiple parameters as a group with spread operator

-> ex. function buildName(firstName: string, …restOfName: string[]) {}

(4) this and arrow functions

-> ECMAScript 6 arrow functions syntax captures the ‘this’ where the function is created rather than where it is invoked

-> --noImplicitThis: this flag will warn you about probable mistakes due to wrong this syntax

(5) this parameters

-> If ‘this’ is not typed explicitly, it is always of type ‘any’

-> To fix this, you can provide an explicit this parameter. this parameters are fake parameters that come first in the parameter list of a function

ex. 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),

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 };

}

}

(6) this parameters in callbacks

-> You can also run into errors with this in callbacks, when you pass functions to a library that will later call them. Because the library that calls your callback will call it like a normal function, this will be undefined

-> First, the library author needs to annotate the callback type with this. this: void means that addClickListener expects onclick to be a function that does not require a this type

-> ex. interface UIElement {

addClickListener(onClick: (this: void, e: Event) => void): void;

}

class Handler {

info: string;

onClickBad(this: void, e: Event) {}

}

let h = new Handler();

uiElement.addClickListener(h.onClickBad);

-> With this method, since ‘this’ type is void, we cannot use ‘this’ inside the code such as ‘this.info’. If we want it, we’ll have to use an arrow function

-> ex. class Handler {

info: string;

onClickGood = (e: Event) => { this.info = e.message }

}

-> This works because arrow functions don’t capture this, so we can always pass them to something that expects this: void. The downside is that one arrow function is created per object of type Handler. Methods, on the other hands, are only created once and attached to Handler’s prototype. They are shared between all objects of type Handler

(7) Overloads

-> Supply multiple function types for the same function as a list of overloads

-> ex. function pickCard(x: { suit: string; card: number; }[]): number;

function pickCard(x: number): { suit: string; card: number; };

function pickCard(x): any { … } // This line is necessary

6. Generics

(1) Generic Type Variables, Generic Types

-> type variable: a special kind of variable that works on types rather than values

-> ex. function identity<T>(arg: T): T { return arg; }

-> There are 2 kinds of generic function

-> ex. let output = identity<string>(“myString”); // explicitly set T

-> ex. let output = identity(“myString”); // type argument inference

-> type argument inference: we want the compiler to set the value of T for us automatically based on the type of the argument we pass in

-> ex. let myIdentity: <T>(arg: T) => T = identity;

let myIdentity2: <U>(arg: U) => U = identity; // We could also have used a different name for the generic type parameter in the type, so long as the number of type variables and how the type variables are used line up

-> ex. interface GenericIdentityFn {

<T>(arg: T): T;

}

function identity<T>(arg: T): T { return arg; }

let myIdentity: GenericIdentityFn = identity;

-> ex. interface GenericIdentityFn<T> {

(arg: T): T;

}

let myIdentity = GenericIdentityFn<number> = identity;

-> It is not possible to create generic enums and namespaces

-> We can also write the generic type as a call signature of an object literal type

-> ex. let myIdentity: {<T>(arg:T): T} = identity;

(2) Generic Classes

-> Generic classes are only generic over their instance side than their static side, so when working with classes, static members can not use the class’s type parameter

(3) Generic Constraint

-> ex. interface Lengthwise {   
 length: number;

}

function loggingIdentity<T extends Lengthwise>(arg: T): T {

console.log(arg.length);

return arg;

}

loggingIdentity(3); // Error, number doesn’t have a .length property

loggingIdentity({length: 10, value: 3}) // OK

-> ex. 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”); // OK

getProperty(x, “m”); // Error

(4) Using Class Types in Generics

-> When creating factories in TypeScript using generics, it is necessary to refer to class types by their constructor functions

-> ex. function create<T>(c: { new(): T; }): T { return new c(); }

-> ex. function create<T>(c: new() => T): T { return new c(); }

-> ex. class BeeKeeper {

hasMask: boolean;

}

class ZooKeeper {

nametag: string;

}

class Animal {

numLegs: number

}

class Bee extends Animal {

keeper: BeeKeper;

}

class Lion extends Animal {

keeper: ZooKeeper;

}

function createInstance<A extends Animal>(c: new() => A): A {

return new c();

}

7. Enums

-> Enum members have numeric value associated with them and can be either constant or computed

-> reverse mapping from enum values to enum names is possible

-> ex. enum Enum { A }

let a = Enum.A;

let nameOfA = Enum[Enum.A];

-> const enums are defined using the const modifier that precedes enum keyword

-> const enums can only use constant enum expressions and unlike regular enums they are completely removed during compilation. Const enum members are inlined at use sites. This is possible since const enums cannot have computed members

//TODO: Why is Ambient enums needed?

-> Ambient enums: used to describe the shape of already existing enum types

-> Compiler does not emit the codes for ambient enums

-> One important difference between ambient and non-ambient enums is that, in regular enums, members that don’t have an initializer are considered constant members. For non-const ambient enums member that does not have initializer is considered computed

8. Type Inference

-> Type inference takes place when initializing variables and members, setting parameter default values and determining function return types

-> Contextual typing occurs when the type of an expression is implied by its location

-> ex. window.onmousedown = function(mouseEvent) {

console.log(mouseEvent.buton); // Error

}

-> TypeScript type checker used the type of the window.onmousedown function to infer the type of the function expression on the right hand side of the assignment. When it did so, it was able to infer the type of the mouseEvent parameter. if this function expression were not in a contextually typed position, the mouseEvent parameter would have type any

-> If the contextually typed expression contains explicit type information, the contextual type is ignored

-> ex. window.onmousedown = function(mouseEvent: any)

console.log(mouseEvent.buton); // Now, no error is given

9. Type Compatibility

-> structural typing: way of relating types based solely on their members

-> ex. interface Named { name: string; }

class Person { name: string; }

let p: Named;

p = new Person(); // OK

-> The basic rule for TypeScript’s structural type system is that x is compatible with y if y has at least the same members as x

-> x = y: To check whether y can be assigned to x, the compiler checks each property of x to find a corresponding compatible property in y

-> ex. interface Named { name: string; }

let x : Named;

let y = { name: “Alice”, location: “Seattle” };

x = y; // OK

-> While comparing primitive types and object types is relatively straightforward, the question of what kinds of functions should be considered compatible is a bit more involved

-> ex. let x = (a: number) => 0;

let y = (b: number, s: string) => 0;

y = x; // OK

x = y; // Error

-> You may be wondering why we allow ‘discarding’ parameters like in the example y = x. The reason for this assignment to be allowed is that ignoring extra function parameters is actually quite common in javascript

-> ex. let x = () => ({name: “Alice”});

let y = () => ({name: “Alice”, location: “Seattle”});

x = y; // OK

y = x // Error

-> When a function has overloads, each overload in the source type must be matched by a compatible signature on the target type

-> Extra optional parameters of the source type are not an error

-> ex. let x = (a: number) => 0;

let y = (a: number, b?: string) => 0

x = y; // OK

(1) Enums Compatibility

-> Enums are compatible with numbers, and numbers are compatible with enums. Enum values from different enum types are considered incompatible

-> ex. enum Status { Ready, Waiting };

enum Color { Red, Blue, Green };

let status = Status.Ready;

status = Color.green // Error

(2) Class Compatibility

-> Classes work similarly to object literal types and interfaces with one exception: they have both a static and an instance type. When comparing two objects of a class type, only members of the instance are compared. Static members and constructors do not affect compatibility

-> ex. class Animal {

feet : number;

constructor(name: string, numFeet: number) {}

}

class Size {  
 feet: number;

constructor(numFeet: number) {}

}

let a: Animal;

let s: Size;

a = s; // OK

s = a; // OK

-> Private and protected members in a class affect their compatibility. When an instance of a class is checked for compatibility, if the target type contains a private member, then the source type must also contain a private member that originated from the same class. Likewise, the same applies for an instance with a protected member

(3) Generics Compatibility

-> Because a TypeScript is a structural type system, type parameters only affect the resulting type when consumed as part of the type of a member. A generic type that has its type arguments specified acts just like a non-generic type(specified: NotEmpty<number>, unspecified: NotEmpty)

-> ex. interface Empty<T> { }

let x: Empty<number>;

let y: Empty<string>;

x = y; // OK

-> ex. interface NotEmpty<T> { data: T }

let x: NotEmpty<number>;

let y: NotEmpty<string>;

x = y; // Error

-> For generic types that do not have their type arguments specified, compatibility is checked by specifying ‘any’ in place of all unspecified type arguments. The resulting types are then checked for compatibility, just as in the non-generic case

-> ex. let identity = function<T>(x: T): T {}

let reverse = function<U>(x: U): U {}

identity = reverse; // OK, because (x:any)=>any matches (y:any)=>any

-> TODO: Subtype vs. Assignment

10. Advanced Types

(1) Intersection Types

-> An intersection type combines multiple types into one. This allows you to add together existing types to get a single type that has all the features you need

-> For example, Person & Serializable & Loggable is a Person and Serializable and Loggable. That means an object of this type will have all members of all three types

(2) Union Types

-> If we have a value that has a union type, we can only access members that are common to all types in the union

-> ex. interface Bird { fly(); layEggs(); }

interface Fish { swim(); layEggs(); }

function getSmallPet(): Fish | Bird {}

let pet = getSmallPet();

pet.layEggs(); // OK

pet.swim(); // Error

(3) Type Guards and Differentiating Types

-> ex. let pet = getSmallPet();

if(pet.swim) { pet.swim(); }

else if(pet.fly) { pet.fly(); }

-> Each of these property accesses will cause an error because as we mentioned, we can only access members that are guaranteed to be in all the constituents of a union type

-> ex. let pet = getSmallPet();

if((<Fish>pet).swim) { (<Fish>pet).swim(); }

else { (<Bird>pet).fly(); } // OK

(4) User-Defined Type Guards

-> A type guard is some expression that performs a runtime check that guarantees they type in some scope. To define a type guard, we simply need to define a function whose return type is a type predicate. A predicate takes the form ‘parameterName is Type’, where parameterName must be the name of a parameter from the current function signature

-> ex. function isFish(pet: Fish | Bird): **pet is Fish** {  
 return (<Fish>pet).swim !== undefined;}

-> Any time isFish is called with some variable, TypeScript will narrow that variable to that specific type if the original type is compatible

-> ex. if(isFish(pet)) { pet.swim(); }

else { pet.fly(); } // Both calls to ‘swim’ and ‘fly’ are now okay

-> Notice that TypeScript not only knows that pet is a Fish in the if branch but it also knows that in the else branch, you don’t have a Fish, so you must have a Bird

(5) typeof Type Guards

-> TypeScript will recognize ‘typeof’ as a type guard on its own

-> ex. function padLeft(value: string, padding: string | number) {

if(typeof padding === “number”) {}

if(typeof padding === “string”) {}

}

-> typeof Type Guards are recognized in two different forms: ‘typeof v === “typename”’ and ‘typeof v !== “typename”’, where “typename” must be “number”, “string”, “boolean”, “undefined”, “object”, “function”, or “symbol”.

(6) instanceof Type Guards

-> instanceof Type Guards are a way of narrowing types using their constructor function

-> ex. class A {}

let a = new A();

a instanceof A // True

-> The right side of the instanceof needs to be a constructor function and TypeScript will narrow down to: // TODO: understand this

1. the type of the function’s prototype property if its type is not any

2. the union of types returned by that type’s construct signatures

(7) Nullable Types

-> --strictNullChecks: when you declare a variable, it doesn’t automatically include null or undefined. You can include them explicitly using a union type. With this flag, an optional parameter(property) automatically adds ‘| undefined’

-> ex. function f(x: number, y?: number) { }

f(1, null); // Error, ‘null’ is not assignable to ‘number | undefined’

-> ex. class C { a: number; b?: number }

let c = new C();

c.a = undefined; // Error, ‘undefined’ is not assignable to ‘number’

c.b = undefined; // OK

c.b = null; // Error, ‘null’ is not assignable to ‘number | undefined’

-> type assertion operator: in cases where the compiler can’t eliminate null or undefined, you can use the type assertion operator to manually remove them. The syntax is **postfix !: identifier!** removes null and undefined from the type of identifier

//TODO: understand this

-> ex. function broken(name: string | null): string {  
 function postfix(epithet: string) {  
 return name.charAt(0) // Error, ‘name’ is possibly null

}

name = name || “Bob”;

return postfix(“great”);

}

function fixed(name: string | null): string {  
 function postfix(epithet: string) {  
 return name!.chartAt(0); // OK

}

name = name || “Bob”;

return postfix(“great”);  
 }

(8) Type Aliases

ex. type Name = string;

type NameResolver = () => string;

type NameOrResolver = Name | NameResolver;

function getName(n: NameOrResolver): Name {  
 if(typeof n === “string”) { return n; }

else { return n(); }

}

-> Just like interfaces, type aliases can also be generic

-> ex. type LinkedList<T> = T & { next: LinkedList<T> };

interface Person { name: string; }

var people: LinkedList<Person>;

-> ex. type Tree<T> = { value: T; left: Tree<T>; right: Tree<T> }

-> Interfaces vs. Type Aliases

1. interfaces create a new name that is used everywhere. Type aliased don’t create a new name – for instance, error messages won’t use the alias name. In the code below, hovering over ‘interfaced’ in an editor will show that it returns an ‘Interface’ but will show that ‘aliased’ returns object literal type

-> ex. type Alias = { num: number }  
 interface Interface { num: number }

declare function aliased(arg: Alias): Alias;

declare function interfaced(arg: Interface): Interface;

2. aliases cannot be extended or implemented from (nor can they extend/implement other types). In this case, you should always use an interface over a type alias if possible

(9) String Literal Types

-> In practice, string literal types combine nicely with union types, type guards, and type aliases. You can use these features together to get enum-like behavior with strings

-> ex. type Easing = “ease-in” | “ease-out” | “ease-in-out”;

class UIElement {

animate(dx: number, dy: number, easing: Easing) {

if(easing === “ease-in”) {}

else if(easing === “ease-out”) {}

else if(easing === “ease-in-out”)

}

}

let button = new UIElement();

button.animate(0, 0, “ease-in”);

button.animate(0, 0, “uneasy”); // Error: “uneasy” is not assignable to parameter of type ‘”ease-in” | “ease-out” | “ease-in-out”’

-> String literal types can be used in the same way to distinguish overloads

-> ex. function a(b: “img”);

function a(b: “input”);

function a(b: string) {}

(10) Discriminated Unions

-> You can combine string literal types, type guards and type aliases to build an advanced pattern called discriminated unions, also known as tagged unions or algebraic data types

-> ex. interface Square { kind: “square”; size: number; }

interface Rectangle {

kind: “rectangle”;

width: number;

height: number;

}

interface Circle { kind: “circle”; radius: number; }

type Shape = Square | Rectangle | Circle;

function area(s: Shape) {

switch(s.kind) {

case “square”: return s.size \* s.size;

case “rectangle”: return s.height \* s.width;

case “circle”: return Math.PI \* s.radius \*\* 2;

}

}

-> Exhaustiveness checking: we would like the compiler to tell us when we don’t cover all variant of the discriminated union. For example, if we add Triangle to Shape, we need to update area as well. There are two ways

-> ex1. turn on –strictNullChecks and specify a return type

function area(s: Shape): number { … } // Error: return type has to be number | undefined

-> Because the switch statement is no longer exhaustive, TypeScript is aware that the function could sometimes return undefined

-> ex2. use the ‘never’ type that the compiler uses to check for exhaustiveness // TODO: why is the type of s is ‘never’?

function assertNever(x: never): never {

throw new Error(“Unexpected object: “ + x);

}  
 function area(s: Shape) {

switch(s.kind) { // Above the ‘default’ code is the same as before

default: return assertNever(s); }}

(11) Polymorphic this types

-> // TODO: fully understand the documentation

-> **this-types**: Every class and interface has a ***this-type*** that represents the actual type of instances of the class or interface within the declaration of the class or interface. The this-type is referenced using the keyword **this** in a type position. Within instance methods and constructors of a class, the type of the expression **this** is the this-type of the class.

Classes and interfaces support inheritance and therefore the instance represented by **this** in a method isn't necessarily an instance of the containing class—it may in fact be an instance of a derived class or interface. To model this relationship, the this-type of a class or interface is classified as a type parameter. Unlike other type parameters, it is not possible to explicitly pass a type argument for a this-type. Instead, in a type reference to a class or interface type, the type reference itself is implicitly passed as a type argument for the this-type

-> ex. class BasicCalculator {

public constructor(protected value: number = 0) {}  
 public currentValue() { return this.value; }

public add(operand: number): this {

this.value += operand;

return this;

}

public multiply(operand: number): this {  
 this.value \*= operand;

return this;

}

}

class ScientificCalculator extends BasicCalculator {

public constructor(value = 0) { super(value); }  
 public sin() {

this.value = Math.sin(this.value);

return this;

}  
 }

let v = new BasicCalculator(2).multiply(5).add(1);

let s = new ScientificCalculator(2).multiply(5).sin();

(12) Index Types

-> ex. function pluck<T, K extends **keyof** T>(o: T, names: K[]): **T[K]**[] {

return names.map(n => o[n]);

}

interface Person { name: string; age: number; }

let person: Person = { name: ‘Jarid’, age: 35; };

let strings: string[] = pluck(person, [‘name’]);

-> **keyof:** the index type query operator. For any type T, **keyof T** is the union of known, public property names of T

-> ex. interface Person { name: string; age: number; }

let personProps: keyof Person; // ‘name’ | ‘age’

-> **T[K]**: the indexed access operator. Here, the type syntax reflects the expression syntax. That means that person[‘name’] has the type Person[‘name’]

-> Once you return the T[K] result, the compiler will instantiate the actual type of they key so the return type of getProperty will vary according to which property you request

-> ex. function getProperty<T, K extends keyof T>(o: T, name: K): T[K] {

return o[name];

}

let name: **string** = getProperty(person, ‘name’);

let age: **number** = getProperty(person, ‘age’);

let unknown = getProperty(person, ‘unknown’); // Error

-> keyof and T[K] interact with string index signatures. If you have a type with a string index signature, keyof T will just be string. And T[string] is just the type of the index signature

-> ex. interface Map<T> { [key: string]: T; }

let keys: keyof Map<number>; // string

let value: Map<number>[‘foo’] // number

(13) Mapped Types

-> TypeScript provides a way to create new types based on old type – mapped types. In a mapped type, the new type transforms each property in the old type in the same way. For example, you can make all properties of a type readonly or optional

-> ex. type ReadOnly<T> = { readonly [P **in** keyof T]: T[P]; }

type Partial<T> = { [P **in** keyof T]?: T[P]; }

type Keys = ‘option1’ | ‘option2’;

type Flags = { [K in Keys]: boolean };

<=> type Flags = { option1: boolean; option2: boolean; }(hardcoded)

-> The syntax resembles the syntax for index signatures with a **for-in** inside

1. The type variable K, which gets bound to each property in turn

2. The string literal union Keys, which contains the names of properties to iterate over

3. The resulting type of the property

-> ex. type Record<K extends string, T> = { [P in K]: T; }

type ThreeStringProps = Record<’prop1’ | ‘prop2’ | ‘prop3’, string>

=> { prop1: string, prop2: string, prop3: string }

-> // TODO: review the concept of homomorphic and unwrapping inference

11. Symbols

-> symbol is a primitive data type

-> symbol values are created by calling the Symbol constructor

-> Symbols are immutable and unique

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

-> ex. 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

-> ex. const getClassNameSymbol = Symbol();

class C {

[getClassNameSymbol]() { return “C” };

}  
 let c = new C();

let className = c[getClassNameSymbol](); // “C”

12. Iterators and Generators

-> for-of: it loops over an iterable object such as Array

-> for-of vs. for-in: both statements iterate over lists. for-in returns a list of keys on the object being iterated, whereas for-of returns a list of values of the numeric properties of the object being iterated

-> Another distinction is that for-in operates on any object. for-of, on the other hand, is mainly interested in values of iterable objects

13. Modules

-> Modules are executed within their own scope, not in the global scope; this means that variables, functions, classes, etc. declared in a module are not visible outside the module unless they are explicitly exported using one of the export forms. Conversely, to consume a variable, function, class, interface, etc. exported from a different module, it has to be imported using one of the import forms

-> Modules import one another using a module loader. At runtime the module loader is responsible for locating and executing all dependencies of a module before executing it

(1) Export

-> Any declaration(such as a variable, function, class, type alias, or interface) can be exported by adding the export keyword

-> Exporting a declaration

-> ex. export interface StringValidator {}

export const number;

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

-> ex. export { StringValidator }

export { number as Number }

-> 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

-> Optionally, a module can wrap one or more modules and combine all their exports using **export \* from “module”** syntax. For example, if there is ‘a.ts’, ‘b.ts’, ‘c.ts’ and ‘all.ts’,

->ex. in “all.ts”,

export \* from “./a.ts”;

export \* from “./b.ts”;

export \* from “./c.ts”;

(2) Import

-> imports can be renamed

-> ex. import { ZipCodeValidator as ZCV } from “./ZipCodeValidator”;

-> Import the entire module into a single variable and use it to access the module exports

-> ex. 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,

-> ex. import “./my-module.js”;

(3) Default exports

-> there can only be one default export per module

-> default exports are imported using a different form: no need of ‘{}’

-> ex. import validator from “./ZipCodeValidator”;

-> ex. export default function(s: string) { return 0; } (in a.ts)

import func from ‘./a’;

console.log(func(‘sdsd’)); // 0 (in test.ts)

-> default exports can also be just values

-> ex. export default “123”; (in a.ts)

import num from ‘./a’;

console.log(num); // 123 (in test.ts)

(4) **export =** and **import = require()**

-> **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 =**, TypesScript-specific **import module = require(“module”)** must be used to import the module

-> ex. (in a.ts)

class A {}

export = A;

(in test.ts)

import zip = require(“./a”);

let a = new A();

(5) Code Generation for Modules

(6) Optional Module Loading

(7) Working with Other JavaScript Libraries

// TODO: See (5), (6), (7)

(8) Guidance for structuring modules

-> Export as close to top-level as possible(not too many levels of nesting)

-> If you’re only exporting a single class or function, use export default

-> Explicitly list imported names

-> ex. import { SomeType, someFunc } from “./MyThings”;

-> If you’re importing a large number of things, use the namespace import pattern

-> ex. (in MyLargeModule.ts)

export class Dog {}  
 export class Cat {}

export class Tree {}

export class Flower {}

(in 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. The recommended solution is to not mutate the original object, but rather export a new entity that provides the new functionality

-> ex. (in Calculator.ts)

export class Calculator {}

export function test(c: Calculator, input: string) {}

(in testCalculator.ts)

import { Calculator, test } from ‘./Calculator’;

let c = new Calculator();

test(c, “1+2\*44/11=”);

(in ProgrammerCalculator.ts)

import { Calculator } from ‘./Calculator’;

class ProgrammerCalculator extends Calculator {}  
 export { ProgrammerCalculator as Calculator };

export { test } from ‘./Calculator’;

(in testProgrammerCalculator.ts)

import { Calculator, test } from ‘./ProgrammerCalculator’;

let c = new Calculator(2);

test(c, “001+010=”);

-> Do not use namespaces in modules

14. Namespaces



-> **namespace namespaceName {}**

-> if we want the interfaces, classes, etc. in namespace to be visible outside the namespace, we preface them with export

-> Outside namespace, exported ones can be accessed with **namespaceName.name**

(1) Splitting Across Files

-> We can split the 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



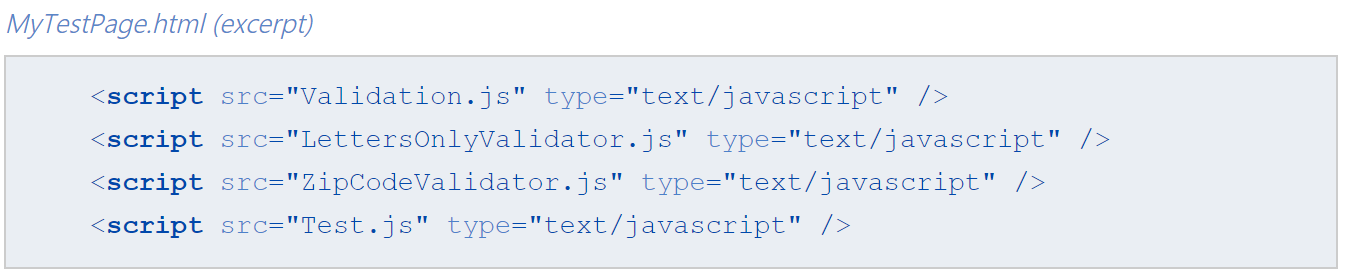


-> Once there are multiple files involved, we’ll need to make sure all of the compiled code gets loaded

1. 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

2. 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 to our webpage to load each emitted file in the appropriate order



(2) Aliases

-> ex. namespace Shapes {  
 export namespace Polygons {

export class Triangle {}

export class Square {}

}  
 }

import polygons = Shapes.Polygons;

let sq = new polygons.Square();

15. Namespaces and Modules

(1) Pitfalls of Namespaces and Modules

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

-> A common mistake is 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 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 .d.ts file

-> ex. (in myModules.d.ts)

declare module “SomeModule” {

export function fn(): string;

}  
(in 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

16. Module Resolution

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

-> 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 or Node. These strategies tell the compiler where to look for A

-> If that didn’t work and if the module name is non-relative, then the compiler will attempt to locate an ambient module declaration

-> Finally, if the compiler could not resolve the module, it will log an error

(1) Relative vs. Non-relative module imports

-> A relative import is one that starts with **/**, **./**, or **../**. A relative import is resolved relative to the importing file and cannot resolve to an ambient module declaration

-> Any other import is considered non-relative. A non-relative import can be resolved relative to baseUrl, or through path mapping. They can also resolve to ambient module declarations

-> ex. import \* as $ from “jqeury”

import { Component } from “@angular/core”

(2) Module Resolution Strategies

-> There are two possible module resolution strategies: Node and Classic. You can use the –moduleResolution flag to specify the module resolution strategy

-> Classic

-> 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/module.ts

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

-> For non-relative import, however, the compiler walks up the directory tree starting with the director containing the importing file, trying to locate a matching definition file

-> ex. **import { b } from “moduleB”** in **/root/src/folder/A.ts**

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: attempts to mimic the Node.js module resolution mechanism at runtime

-> 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 import

-> ex. let’s consider a file located at **/root/src/A.js**, which contains the import **var x = require(“./B”)**. Node.js resolves that import in the following order

1. As the file named /root/src/B.js, if it exists

2. As the folder /root/src/B if it contains a file named package.json that specifies a “main” module. In our example, if Node.js found the file /root/src/B/package.json containing { “main”: “lib/mainModule.js” }, then Node.js will refer to /root/src/B/lib/mainModule.js

3. As the folder /root/src/B if it contains a file named index.js. That file is implicitly considered that folder’s “main” module

-> Non-relative import

-> 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

-> ex. Following up our example above, consider if **/root/src/A.js** instead used a non-relative path and had the import **var x = require(“B”)**. Node would then try to resolve B to each of the locations until one worked

1. /root/src/node\_modules/B.js

2. /root/src/node\_modules/B/package.json (if specifies a “main” property)

3. /root/src/node\_modules/B/index.js

4. /root/node\_modules/B.js

5. /root/node\_modules/B/package.json (if specifies a “main” property)

6. /node\_modules/B.js

7. /node\_modules/B/package.json (if specifies a “main” property)

8. /node\_modules/B/index.js

-> 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, .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

(3) Additional module resolution flags

// TODO: Understand this

(4) 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 determined as either:

1. value of baseUrl command line argument(if given path is relative, it is computed based on current directory)

2. 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

(5) Path Mapping

-> Sometimes modules are not directly located under baseUrl. Loaders use a mapping configuration to map module names to files at run-time

-> The TypeScript compiler supports the declaration of such mappings using **paths** property in tsconfig.json files

-> ex. {  
 “compilerOptions”: {  
 “baseUrl”: “.”, // This must be specified if ‘paths’ is

“paths”: {

// This mapping is relative to “baseUrl”

“jquery”: [“node\_modules/jquery/dist/jquery”]

}

}  
 }



-> This tells the compiler for any module import that matches the pattern “\*”(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:

1. import ‘folder1/file2’

=> pattern “\*” is matched and wildcard captures the whole module name

=> try first substitution in the list: “\*” -> folder1/file2

=> result of substitution is non-relative name – combine it with baseUrl -> projectRoot/folder1/file2.ts

=> File exists. Done

2. import ‘folder2/file3’

=> pattern “\*” is matched and wildcard captures the whole module name

=> try first substitution in the list: “\*” -> folder2/file3

=> result of substitution is non-relative name – combine it with baseUrl -> projectRoot/folder2/file3.ts

=> File does not exist, move to the second substitution

=> second substitution ‘generated/\*’ -> generated/folder2/file3

=> result of substitution is non-relative name – combine it with baseUrl -> projectRoot/generated/folder2/file3.ts

=> File exist. Done

(6) 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 imform 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



-> 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

-> ex. {

“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

-> // TODO: fully understand this

-> ex. 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}/message. In this hypothetical setup the tool enumerates supported locales, mapping the abstracted path into ./zh/messages, ./de/messages, and so forth

-> 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

-> {

“compilerOptions”: {  
 “rootDirs”: [

“src/zh”, “src/de”, “src/#{locale}

]

}  
 }

-> Tracing module resolution: use flag **–traceResolution**

-> 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.

17. Declaration Merging

-> 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

-> In TypeScript, a declaration creates entities in at least one of three groups: namespace, type or value



(1) Merging Interfaces

-> the merge mechanically joins the members of both declarations into a single interface with the same name

-> ex. 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. In the case of interface A merging with later interface A, the second interface will have a higher precedence than the first

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

(2) Merging Namespaces

-> 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

-> 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

-> ex. namespace Animals {

let haveMuscles = true;

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 {}

}

-> Because haveMuscles is not exported, only the Zebra function that shares the same un-merged namespace can see the symbol. The Dog function, even though it’s part of the merged Animals namespace cannot see this un-exported number

-> 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

-> ex. class Album {  
label: Album.AlbumLabel;

}

namespace Album {

export class AlbumLabel{ }

export var a = 5; // static member of class Album

}

-> ex. var Album = (function () {

function Album() {}

return Album;

}());

(function (Album) {

var AlbumLabel = (function () {

function AlbumLabel() {}

return AlbumLabel;

}());

Album.AlbumLabel = AlbumLabel;

Album.a = 5;

})(Album || (Album = {}));

-> This gives the user a way of describing inner classes. The visibility rules for merged members is the same as described in the ‘Merging Namespaces’ section, so we must export the AlbumLabel class for 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

-> ex. function builLabel(name: string): string {  
 return buildLabel.prefix + name + buildLabel.suffix;

}  
 namespace buildLabel {  
 export let suffix = “”;

export let prefix = “Hello, “;

}

-> Similarly, namespaces can be used to extend enums static members

-> ex. 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;

}

}

} // mixColor function is now static member of enum Color

-> Classes cannot merge with other classes or with variables

(3) Module Augmentation

-> // TODO: Understand this

-> Although JavaScript modules do not support merging, you can patch existing objects by importing and then updating them