**1. Object destructing 1**

**2. Property renaming 1**

**3. Default values 1**

**4. Spread Array 1**

**5. Spread Object 2**

**6. Readonly modifier 3**

**7. Accessors 3**

**8. Static Properties 4**

**9. Abstract Classes 4**

**10. Constructor functions 5**

**11. Functions 6**

**12. Inferring the types 6**

**13. Optional and Default Parameters 6**

**14. Rest Parameters 7**

**15 call(this) 7**

**16. apply(this) 8**

**17. this and arrow functions8**

function f([f, s]: [number, number]): void { }

f([2, 3])

**-----------Object destructing ---------------**

let o = { a: "a", b: 2, c: "c" };

let { a , b } = o; //output a 2

({ a, b } = { a: "abc", b: 101 })

let { a, ...passthrough } = o;

let total = passthrough.b+passthrough.c.lle

**-----------Property renaming ---------------**

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

let newName1 = o.a;

let newName2 = o.b;

let { a, b }: { a: string, b: number } = o;

**-----------Default values ---------------**

let { a, b = 100 } = { a: "a" };

function keepWholeObject(wholeObject: { a: string, b?: number }) { let { a, b = 100 } = wholeObject; }

keepWholeObject({ a: "a" })

function f({ a, b = 0 } = { a: "" }): { a: string, b: any } { return { a: a, b: b }

}

console.log(f()) //output {a: "", b: 0}

console.log(f({a:"yes"})) //output {a: "yes", b: 0}

console.log(f({})) //output Property 'a' is missing in type '{}'

**-----------Spread ---------------**

**-----------Spread Array ---------------**

let fst = [1, 2]

let scnd = [3, 4]

let bothplush = [0, ...fst, ...scnd]

console.log(bothplush) //output [0, 1, 2, 3, 4]

**-----------Spread Object ---------------**

let defaults = { food: "spicy", price: "20", ambiance: "noisy" }

let search1 = { ...defaults, food: "rich" }

console.log(JSON.stringify(search1)) //output {"food":"rich","price":"20","ambiance":"noisy"}

let search2 = { food: "rich", ...defaults }

console.log(JSON.stringify(search2)) //output {"food":"spicy","price":"20","ambiance":"noisy"}

class C { p = 12;

m() {

console.log("test", this.p)

}

}

let c = new C();

let clone = { ...C }

clone.m() //error

clone.p //error

class Animal {

name: string;

constructor(theName: string) {

this.name = theName;

}

move(distanceInMeters: number) {

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

}

}

class Snake extends Animal {

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

move(distanceInMeters = 5) {

console.log("Snake...");

super.move(distanceInMeters)

}

}

class Horse extends Animal {

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

move(distanceInMeters = 45) {

console.log("Horse...")

super.move(distanceInMeters)

}

}

// let sam = new Snake("Sammy the Python")

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

// sam.move();

// tom.move(34)

**---------Readonly modifier--------**

Readonly properties must be initialized at their declaration or in the constructor.

class Octopus {

readonly name: string;

readonly numberOfLegs: number = 8;

constructor (theName: string) {

this.name = theName;

}

}

let dad = new Octopus("Man with the 8 strong legs");

console.log(dad.name)

dad.name = "Man with the 3-piece suit"; // error! name is readonly.

Parameter properties let you create and initialize a member in one place.

class Octopus {

constructor(readonly name: string) { //We’ve consolidated the declarations and assignment into one location.

} }

let dad = new Octopus("Man with the 8 strong legs");

**// ---------Accessors--------**

**This gives you a way of having finer-grained control over how a member is accessed on each object.**

class Employee {

fullName: string;

}

let employee = new Employee();

employee.fullName = "Bob Smith";

if (employee.fullName) {

console.log(employee.fullName);

}

While allowing people to randomly set fullName directly is pretty handy, we may also want enforce some constraints when fullName is set.

const fullNameMaxLength = 10;

class Employee {

private \_fullName: string;

get fullName(): string {

return this.\_fullName;

}

set fullName(newName: string) {

if (newName && newName.length > fullNameMaxLength) {

throw new Error("fullName has a max length of " + fullNameMaxLength);

}

this.\_fullName = newName;

}

}

let employee = new Employee();

employee.fullName = "Bob Smith";

if (employee.fullName) {

console.log(employee.fullName);

}

**---------Static Properties--------**

We can also create static members of a class, those that are visible on the class itself rather than on the instances.

class Grid {

static origin = {x: 0, y: 0};

calculateDistanceFromOrigin(point: {x: number; y: number;}) {

let xDist = (point.x - Grid.origin.x);

let yDist = (point.y - Grid.origin.y);

return Math.sqrt(xDist \* xDist + yDist \* yDist) / this.scale;

}

constructor (public scale: number) { }

}

let grid1 = new Grid(1.0); // 1x scale

let grid2 = new Grid(5.0); // 5x scale

console.log(grid1.calculateDistanceFromOrigin({x: 10, y: 10})); console.log(grid2.calculateDistanceFromOrigin({x: 10, y: 10}));

**---------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.

abstract class Department {

constructor(public name: string) { }

printName(): void {

console.log("Department name: " + this.name);

}

abstract printMeeting(): void; // must be implemented in derived classes

}

class AccountingDepartment extends Department {

constructor() {

super("Accounting and Auditing"); // constructors in derived classes must call super()

}

printMeeting(): void {

console.log("The Accounting Department meets each Monday at 10am.");

}

generateReports(): void {

console.log("Generating accounting reports...");

}

}

let department: Department; // ok to create a reference to an abstract type

department = new Department(); // error: cannot create an instance of an abstract class

department = new AccountingDepartment(); // ok to create and assign a non-abstract subclass

department.printName();

department.printMeeting();

department.generateReports(); // error: method doesn't exist on declared abstract type

**---------Constructor functions--------**

class Greeter { }

let greeter: Greeter;

greeter = new Greeter();

let greeter1: Greeter = new Greeter();

let greeterMaker: typeof Greeter = Greeter;

**---------Functions--------**

Functions are the fundamental building block of any application in JavaScript.

TypeScript functions can be created both as a named function or as an anonymous function

Named function

function add(x: number, y: number) {

return x + y;

}

console.log(add(4, 5))

Anonymous function

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

console.log(myAdd(6, 7))

//\ ---Writing the function type

//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.

// let myAdd: (x: number, y: number) => number =

// function (x: number, y: number): number { return x + y; };

// console.log(myAdd(5, 4))

**---------Inferring the types --------**

TypeScript compiler can figure out the type even if you only have types on one side of the equation:

// myAdd has the full function type

// let myAdd = function (x: number, y: number): number { return x + y; };

// console.log(myAdd(10, 23))

// // The parameters 'x' and 'y' have the type number

// let myAdd2: (baseValue: number, increment: number) => number =

// function (x, y) { return x + y; };

// console.log(myAdd(13, 23))

**---------Optional and Default Parameters --------**

parameter is assumed to be required by the function, the number of arguments given to a function has to match the number of parameters the function expects.

function buildName(firstName: string, middleName?: string, lastName: string = "Islam") {

return `${firstName} ${middleName?middleName:''} ${lastName}`;

}

let result1 = buildName("Bob");

console.log(result1);

let result2 = buildName("Bob", "Adams", "Sr.");

console.log(result2);

let result3 = buildName("Bob", "Adams");

console.log(result3);

let result4 = buildName("Bob", "Adams", "Sr.", "John"); // error, too many parameters

console.log(result4);

**---------Rest Parameters --------**

Rest parameters are treated as a boundless number of optional parameters. When passing arguments for a rest parameter, you can use as many as you want; you can even pass none.

function buildName(firstName: string, ...restOfName: string[]) {

return firstName + " " + restOfName.join(" ");

}

let employeeName = buildName("Joseph", "Samuel", "Lucas", "MacKinzie");

let buildNameFun: (fname: string, ...rest: string[]) => string = buildName;

console.log(buildNameFun("Joseph", "Samuel", "Lucas", "MacKinzie"))

**---------call(this) --------**

function hello(thing) {

console.log(**this** + " says hello " + thing);

}

The call() method is a predefined JavaScript method.

With the call() method, you can write a method that can be used on different objects.

The **call()** method calls a function with a given this value and arguments provided individually.

With call(), an object can use a method belonging to another object.

hello.call("**Yehuda**", "world")

function hello(thing) {

console.log("Hello " + thing);

}

this: hello("world")

desugars to: hello.call(window, "world");

This behavior has changed in ECMAScript 5 only when using strict mode[2]:

this: hello("world")

desugars to: hello.call(undefined, "world");

// a function invocation like fn(...args) is the same as fn.call(window [ES5-strict: undefined], ...args).

var person = {  
  fullName: function(city, country) {  
    return this.firstName + " " + this.lastName + "," + city + "," + country;  
  }  
}  
var person1 = {  
  firstName:"John",  
  lastName: "Doe"  
}  
person.fullName.call(person1, "Oslo", "Norway");

**---------apply(this) --------**

The apply() method accepts arguments in an array:

var person = {  
  fullName: function(city, country) {  
    return this.firstName + " " + this.lastName + "," + city + "," + country;  
  }  
}  
var person1 = {  
  firstName:"John",  
  lastName: "Doe"  
}  
person.fullName.apply(person1, ["Oslo", "Norway"]);

**-----------this and arrow functions------------**

In JavaScript, this is a variable that’s set when a function is called.

let deck = {

suits: ["hearts", "spades", "clubs", "diamonds"],

cards: Array(52),

createCardPicker: function () {

this.suits[0]

return function () {

let pickedCard = Math.floor(Math.random() \* 52);

let pickedSuit = Math.floor(pickedCard / 13);

return { suit: this.suits[pickedSuit], card: pickedCard % 13 }; // error because this being used in the function created by createCardPicker will be set to window instead of our deck object.

A top-level non-method syntax call like this will use window for this. (Note: under strict mode, this will be undefined rather than window).

}

}

}

let cardPicker = deck.createCardPicker();

let pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);

//TypeScript will warn you when you make this mistake if you pass the --**noImplicitThis** flag to the compiler.

//---\*\*We can fix this by making sure the function is bound to the correct this

//before we return the function to be used later.

//This way, regardless of how it’s later used, it will still be able to see the original deck object.

**To do this, we change the function expression to use the ECMAScript 6 arrow syntax.**

let deck = {

    suits: ["hearts", "spades", "clubs", "diamonds"],

    cards: Array(52),

    createCardPicker: function() {

*//allowing us to capture 'this' right here*

*// Arrow functions capture the this where the function is created rather than where it is invoked:*

        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);

**-----------this parameters ------------**

this parameters are fake parameters that come first in the parameter list of a function

interface Deck {

    suits: string;

    createCardPicker(this: Deck)*: () => {}*;

}

let deck = {

    suits: "test",

*// NOTE: The function now explicitly specifies that its callee must be of type Deck*

    createCardPicker: function (this: Deck) {

        return () => {

            return this.suits

        }

    }

}

let cardPicker = deck.createCardPicker()();

console.log(cardPicker);

this is of type Deck now, not any, so --**noImplicitThis** will not cause any errors

**-----------this parameters in callbacks ------------**

//\*\*\*\*\*\*\*\*will learn in future\*\*\*\*\*\*\*\*

//-----------Overloads--------

//\*\*\*\*\*\*\*\*will learn in future\*\*\*\*\*\*\*\*

**----\*\*\*-------Generics----\*\*\*----**

The identity function is a function that will return back whatever is passed in.

function getAge<T>(age: T)*: T*{

    return age;

}

let age = getAge<string>("10")

Here we explicitly set T to be string as one of the arguments to the function call

 class person {

     firstname: string;

     lastname: string;

     constructor(*private* firstName: string, *private* lastName: string) {

         this.firstname = firstName;

         this.lastname = lastName;

     }

 }

 class admin *extends* person { }

 class manage *extends* person { }

 let adminObj = new admin("f", "l");

 let personObj = new person("f", "l");

 function personEcho<T *extends* person>(person: T)*: T*{

     return person;

 }

 let foo = personEcho(personObj)

 function identity(arg: any)*: any*{

     return arg;

 }

losing the information about what that type was when the function returns

type variable, a special kind of variable that works on types rather than values

function identity<T>(arg: T)*: T*{

    return arg;

}

function loggingIdentity<T>(arg:T)*:T*{

    console.log(arg.length);  *// Error: Property 'length' does not exist on type 'T'.*

    return arg;

}

*//When we do, the compiler will give us an error that we’re using the .length member of arg*

function loggingIdentity<T>(...arg: T[])*: T[]*{

    console.log(arg.length);  *// 5*

    return arg;

}

loggingIdentity(3,4,5,6,7)

function identity<T>(arg: T)*: T*{

    return arg;

}

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

console.log(myIdentity(10))

**---generic interface----**

interface *GenericIdentityFn* {

    <T>(arg: T)*: T*;

}

function identity<T>(arg: T)*: T*{

    return arg;

}

let myIdentity: *GenericIdentityFn* = identity;

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 = 0;

myGenericNumber.add = function(x, y) { return x + y; };

we’d like to constrain this function to work with any and all types that also have the.length property.

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;

}

let value = loggingIdentity({length: 10, value: 3})

**---------Using Type Parameters in Generic Constraints---------**

You can declare a type parameter that is constrained by another type parameter.

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'.*

**----- Using Class Types in Generics ----**

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

function create<T>(c: {new()*: T*; })*: T*{

    return new c();

}

class aa{

    constructor(){}

}

let getObjc = create(aa)

console.log(getObjc)

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** **---------------**

Enums allow us to define a set of named constants.

enum Direction {

    Up = 1,

    Down,

    Left,

    Right,

}

console.log(Direction.Down) *//output 2*

members are auto-incremented

enum Direction {

    Up,

    Down,

    Left,

    Right,

}

console.log(Direction.Down) *//output 1*

enum Responses {

    No = 0,

    Yes = 1,

}

function respond(recipient: string, message: Responses)*: void*{

    console.log(recipient, message) *//output Princess Caroline 1*

}

respond("Princess Caroline", Responses.Yes)

following isn’t allowed

enum E {

    A = "test",

    B, *// Error! Enum member must have initializer.*

}

Heterogeneous enums

enum BooleanLikeHeterogeneousEnum {

    No = 0,

    Yes = "YES",

}

**-----Computed and constant members----**

1. a literal enum expression (basically a string literal or a numeric literal)

2. a reference to previously defined constant enum member (which can originate from a different enum)

3. a parenthesized constant enum expression

4. one of the +, -, ~ unary operators applied to constant enum expression

5. +, -, \*, /, %, <<, >>, >>>, &, |, ^ binary operators with constant enum expressions as operands

enum FileAccess {

*// constant members*

    None,

    Read    = 1 << 1,

    Write   = 1 << 2,

    ReadWrite  = Read | Write,

*// computed member*

    G = "123".length

}

console.log(FileAccess.None)  *//output 0*

console.log(FileAccess.Read)   *//output 2*

console.log(FileAccess.Write)   *//output 4*

console.log(FileAccess.ReadWrite)   *//output 6*

console.log(FileAccess.G)   *//output 3*

**----const enums ---**

To avoid paying the cost of extra generated code and additional indirection when accessing enum values, it’s possible to use const enums.

*const* enum Enum {

    A = 1,

    B = A \* 2

}

console.log(Enum.B)  *//output 2*

**-----Ambient enums-------**

Ambient enums are used to describe the shape of already existing enum types.

*declare* enum Enum {

    A = 1,

    B,

    C = 2

}

**--------------Type Inference----------**

type inference is used to provide type information when there is no explicit type annotation.

let x = 3;

The type of the x variable is inferred to be number. This kind of inference takes place when initializing variables and members, setting parameter default values, and determining function return types.

let x = [0, 1, null];

the types of those expressions are used to calculate a “best common type”

two choices for the type of the array: number and null.

function createZoo()*: Animal[]*{

    return [new Rhino(), new Elephant(), new Snake()];

}

In this example, best common type has a set of four candidates: Animal, Rhino, Elephant, and Snake. Of these, Animal can be chosen by the best common type algorithm.

**-----------Type Compatibility----------**

interface Named {

    name: string;

}

let x: Named;

*// y's inferred type is { name: string; location: string; }*

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

x = y;

Note that y has an extra location property, but this does not create an error.

Comparing two function

let x = (a: number) => 0;

let y = (b: number, s: string) => 0;

y = x; *// OK*

x = y; *// Error*

let items = [1, 2, 3];

Don't force these extra parameters

items.forEach((item, index, array) => console.log(item, index, array);

Should be OK!

items.forEach(item => console.log(item));

// Classes

// Static members and constructors do not affect compatibility.

// 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

**-----------Intersection Types--------------**

An intersection type combines multiple types into one.

function extend<First, Second>(first: First, second: Second)*: First & Second*{

    const result: *Partial*<First & Second> = {};

    for (const prop in first) {

        if (first.hasOwnProperty(prop)) {

            (result as First)[prop] = first[prop];

        }

    }

    for (const prop in second) {

        if (second.hasOwnProperty(prop)) {

            (result as Second)[prop] = second[prop];

        }

    }

    return result as First & Second;

}

class Person {

    constructor(*public* name: string) { }

}

interface *Loggable* {

    log(name: string)*: void*;

}

class ConsoleLogger *implements* *Loggable* {

    log(name) {

        console.log(`Hello, I'm ${name}.`);

    }

}

const jim = extend(new Person('Jim'), ConsoleLogger.prototype);

jim.log(jim.name);

**Union Types**

// Union types are closely related to intersection types, but they are used very differently. Occasionally, you’ll run into a library that expects a parameter to be either a number or a string.

/\*\*

\* Takes a string and adds "padding" to the left.

\* If 'padding' is a string, then 'padding' is appended to the left side.

\* If 'padding' is a number, then that number of spaces is added to the left side.

\*/

// function padLeft(value: string, padding: any) {

// Instead of any, we can use a union type for the padding parameter:

// function padLeft(value: string, padding: string | number) {

// if (typeof padding === "number") {

// return Array(padding + 1).join(" ") + value;

// }

// if (typeof padding === "string") {

// return padding + value;

// }

// throw new Error(`Expected string or number, got '${padding}'.`);

// }

// // let indentedString = padLeft("Hello world", 4); // returns " Hello world"

// let indentedString = padLeft("Hello world", true); // errors during compilation

// console.log(indentedString)

// A union type describes a value that can be one of several types. We use the vertical bar (|) to separate each type, so number | string | boolean is the type of a value that can be a number, a string, or a boolean.

// we can only access members that are common to all types in the union.

// interface Bird {

// fly();

// layEggs();

// }

// interface Fish {

// swim();

// layEggs();

// }

// function getSmallPet(): Fish | Bird { }

// let pet = getSmallPet();

// // pet.layEggs(); // okay

// // pet.swim(); // errors

// // -------Type Guards and Differentiating Types---

// let pet = getSmallPet();

// if ((pet as Fish).swim) {

// (pet as Fish).swim();

// } else if ((pet as Bird).fly) {

// (pet as Bird).fly();

// }

// ---User-Defined Type Guards

// A type guard is some expression that performs a runtime check that guarantees the type in some scope.

// function isFish(pet: Fish | Bird): pet is Fish {

// return (pet as Fish).swim !== undefined;

// }

// // Both calls to 'swim' and 'fly' are now okay.

// if (isFish(pet)) {

// pet.swim();

// }

// else {

// pet.fly();

// }

// --------Using the in operator

// The in operator now acts as a narrowing expression for types.

// function move(pet: Fish | Bird) {

// if ("swim" in pet) {

// return pet.swim();

// }

// return pet.fly();

// }

// ---typeof type guards

// function isNumber(x: any): x is number {

// return typeof x === "number";

// }

// console.log(isNumber("10"))

// TypeScript will recognize it as a type guard on its own

// typeof v === "typename", where "typename" must be "number", "string", "boolean", or "symbol".

// instanceof type guards

// instanceof type guards are a way of narrowing types using their constructor function.

// interface Padder {

// getPaddingString(): string

// }

// class SpaceRepeatingPadder implements Padder {

// constructor(private numSpaces: number) { }

// getPaddingString() {

// return Array(this.numSpaces + 1).join(" ");

// }

// }

// class StringPadder implements Padder {

// constructor(private value: string) { }

// getPaddingString() {

// return this.value;

// }

// }

// function getRandomPadder() {

// console.log("called")

// return false ?

// new SpaceRepeatingPadder(4) :

// new StringPadder(" ");

// }

// // Type is 'SpaceRepeatingPadder | StringPadder'

// let padder: Padder = getRandomPadder();

// if (padder instanceof SpaceRepeatingPadder) {

// console.log(padder.getPaddingString()); // type narrowed to 'SpaceRepeatingPadder'

// }

// if (padder instanceof StringPadder) {

// console.log(padder.getPaddingString()); // type narrowed to 'SpaceRepeatingPadder'

// // type narrowed to 'StringPadder'

// }

// ---Nullable types

// TypeScript has two special types, null and undefined

// By default, the type checker considers null and undefined assignable to anything.

// not possible to stop them from being assigned to any type

// The --strictNullChecks flag fixes this: when you declare a variable, it doesn’t automatically include null or undefined. You can include them explicitly using a union type:

// let s = "foo";

// s = null; // error, 'null' is not assignable to 'string'

// let sn: string | null = "bar";

// sn = null; // ok

// sn = undefined; // error, 'undefined' is not assignable to 'string | null'

// -------Optional parameters and properties

// With --strictNullChecks, an optional parameter automatically adds | undefined:

// function f(x: number, y?: number) {

// return x + (y || 0);

// }

// f(1, 2);

// f(1);

// f(1, undefined);

// f(1, null); // error, 'null' is not assignable to 'number | undefined'

// The same is true for optional properties:

// class C {

// a: number;

// b?: number;

// }

// let c = new C();

// c.a = 12;

// c.a = undefined; // error, 'undefined' is not assignable to 'number'

// c.b = 13;

// c.b = undefined; // ok

// c.b = null; // error, 'null' is not assignable to 'number | undefined'

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

// return sn || "default";

// }

// -----Type Aliases

// Type aliases create a new name for a type

// Aliasing doesn’t actually create a new type - it creates a new name to refer to that type.

// type Name = string;

// type NameResolver = () => string;

// type NameOrResolver = Name | NameResolver;

// function getName(n: NameOrResolver): Name {

// if (typeof n === "string") {

// return n;

// }

// else {

// return n();

// }

// }

// We can also have a type alias refer to itself in a property:

// type Tree<T> = {

// value: T;

// left: Tree<T>;

// right: Tree<T>;

// }

// Interfaces vs. Type Aliases

// One difference is that interfaces create a new name that is used everywhere. Type aliases don’t create a new name

// type Alias = { num: number }

// interface Interface {

// num: number;

// }

// declare function aliased(arg: Alias): Alias;

// declare function interfaced(arg: Interface): Interface;

// String Literal Types

// String literal types allow you to specify the exact value a string must have.

// use to get enum-like behavior

//combine nicely with union types, type guards, and type aliases

// 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") {

// }

// else {

// // error! should not pass null or undefined.

// }

// }

// }

// let button = new UIElement();

// button.animate(0, 0, "ease-in");

// button.animate(0, 0, "uneasy"); // error: "uneasy" is not allowed here

// --------Numeric Literal Types

// TypeScript also has numeric literal types.

// function rollDice(): 1 | 2 | 3 | 4 | 5 | 6 {

// // ...

// }

// Discriminated Unions

// You can combine singleton types, union types, type guards, and type aliases to build an advanced pattern called discriminated unions, also known as tagged unions or algebraic data types.

// interface Square {

// kind: "square";

// size: number;

// }

// interface Rectangle {

// kind: "rectangle";

// width: number;

// height: number;

// }

// interface Circle {

// kind: "circle";

// radius: number;

// }

// function assertNever(x: never): never {

// throw new Error("Unexpected object: " + x);

// }

// type Shape = Square | Rectangle | Circle;

// function area(s: Shape): number {

// 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;

// default: return assertNever(s);

// }

// }

// uses the never type that the compiler uses to check for exhaustiveness:

// ---------Symbols------

// symbol is a primitive data type, symbol values are created by calling the Symbol constructor.

// Symbols are immutable, and unique.

// let sym1 = Symbol();

// console.log(sym1.toString())

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

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

// console.log(sym2 === sym3); // false, symbols are unique

// const sym = Symbol();

// let obj = {

// [sym]: "value"

// };

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

// ----------Iterators and Generators----------

// An object is deemed iterable if it has an implementation for the Symbol.iterator property.

// Symbol.iterator function on an object is responsible for returning the list of values to iterate on.

// for..of loops over an iterable object, invoking the Symbol.iterator property on the object.

// let someArray = [1, "string", false];

// for (let entry of someArray) {

// console.log(entry); // 1, "string", false

// }

// for..of => returns a list of values of the numeric properties of the object being iterated, mainly interested in values of iterable objects.

// for..in => returns a list of keys on the object being iterated, operates on any object

// let pets = new Set(["Cat", "Dog", "Hamster"]);

// pets["species"] = "mammals";

// for (let pet in pets) {

// console.log(pet); // "species"

// }

// for (let pet of pets) {

// console.log(pet); // "Cat", "Dog", "Hamster"

// }

// ----------- Modules-------

// Modules are executed within their own scope, not in the global scope;

// 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. Well-known module loaders used in JavaScript are Node.js’s loader for CommonJS modules and the RequireJS loader for AMD modules in Web applications.

// export { ZipCodeValidator as mainValidator };

// export \* from "./StringValidator";

// Import the entire module into a single variable, and use it to access the module exports

// import \* as validator from "./ZipCodeValidator";

// export default "123";

// Namespaces

// 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 Validation.ts LettersOnlyValidator.ts ZipCodeValidator.ts Test.ts

// 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()'

// 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--------

// Using Modules

// Modules can contain both code and declarations.

// Modules also have a dependency on a module loader (such as CommonJs / Require.js) or a runtime which supports ES Modules.

// Modules provide for better code reuse, stronger isolation and better tooling support for bundling.

// Using Namespaces

// Namespaces are a TypeScript-specific way to organize code.

// Namespaces are simply named JavaScript objects in the global namespace. This makes namespaces a very simple construct to use. Unlike modules, they can span multiple files, and can be concatenated using --outFile.

// -----Pitfalls of Namespaces and Modules---

// A common mistake is to try to use the

// /// <reference ... /> syntax to refer to a module file, rather than using an import statement.

// 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.ts file.

// 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.

//----## 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.

// 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

// ------Merging Interfaces

// 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 };

// 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;

// }

// interface Document {

// createElement(tagName: "canvas"): HTMLCanvasElement;

// createElement(tagName: "div"): HTMLDivElement;

// createElement(tagName: "span"): HTMLSpanElement;

// createElement(tagName: string): HTMLElement;

// createElement(tagName: any): Element;

// }

// ------- Decorators------

// 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.

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

// tsc --target ES5 --experimentalDecorators

// tsconfig.json:

// {

// "compilerOptions": {

// "target": "ES5",

// "experimentalDecorators": true

// }

// }

// A Decorator is a special kind of declaration that can be attached to a class declaration, method, accessor, property, or parameter.

// Decorators use the form @expression, where expression must evaluate to a function that will be called at runtime with information about the decorated declaration.

// function sealed(constructor: Function) {

// Object.seal(constructor);

// Object.seal(constructor.prototype);

// }

// @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 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

// class Greeter {

// greeting: string;

// constructor(message: string) {

// this.greeting = message;

// }

// @enumerable(false)

// greet() {

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

// }

// }

// function enumerable(value: boolean) {

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

// descriptor.enumerable = value;

// };

// }

// ----Utility Types

// TypeScript provides several utility types to facilitate common type transformations. These utilities are available globally.

// Partial<T>

// Readonly<T>

// Record<K,T>

// Pick<T,K>

// Omit<T,K>

// Exclude<T,U>

// Extract<T,U>

// NonNullable<T>

// Parameters<T>

// ConstructorParameters<T>

// ReturnType<T>

// InstanceType<T>

// Required<T>

// ThisParameterType

// OmitThisParameter

// ThisType<T>

// Partial<T> Constructs a type with all properties of T set to optional. This utility will return a type that represents all subsets of a given type.

// interface Todo {

// title: string;

// description: string;

// }

// function updateTodo(todo: Todo, fieldsToUpdate: Partial<Todo>) {

// return { ...todo, ...fieldsToUpdate };

// }

// const todo1 = {

// title: 'organize desk',

// description: 'clear clutter',

// };

// const todo2 = updateTodo(todo1, {

// description: 'throw out trash',

// });

// console.log(todo2)

// Readonly<T> Constructs a type with all properties of T set to readonly, meaning the properties of the constructed type cannot be reassigned.

// interface Todo {

// title: string;

// }

// const todo: Readonly<Todo> = {

// title: 'Delete inactive users',

// };

// todo.title = 'Hello'; // Error: cannot reassign a readonly property

// console.log(todo.title)

// Record<K,T> Constructs a type with a set of properties K of type T. This utility can be used to map the properties of a type to another type.

// interface PageInfo {

// title: string;

// }

// type Page = 'home' | 'about' | 'contact';

// const x: Record<Page, PageInfo> = {

// about: { title: 'about' },

// contact: { title: 'contact' },

// home: { title: 'home' },

// };

// Pick<T,K> Constructs a type by picking the set of properties K from T.

// interface Todo {

// title: string;

// description: string;

// completed: boolean;

// }

// type TodoPreview = Pick<Todo, 'title' | 'completed'>;

// const todo: TodoPreview = {

// title: 'Clean room',

// completed: false,

// };

// Omit<T,K> Constructs a type by picking all properties from T and then removing K.

// interface Todo {

// title: string;

// description: string;

// completed: boolean;

// }

// type TodoPreview = Omit<Todo, 'description'>;

// const todo: TodoPreview = {

// title: 'Clean room',

// completed: false,

// };

// Exclude<T,U> Constructs a type by excluding from T all properties that are assignable to U.

// type T0 = Exclude<"a" | "b" | "c", "a">; // "b" | "c"

// type T1 = Exclude<"a" | "b" | "c", "a" | "b">; // "c"

// type T2 = Exclude<string | number | (() => void), Function>; // string | number

// Extract<T,U> Constructs a type by extracting from T all properties that are assignable to U.

// type T0 = Extract<"a" | "b" | "c", "a" | "f">; // "a"

// type T1 = Extract<string | number | (() => void), Function>; // () => void

// NonNullable<T> Constructs a type by excluding null and undefined from T.

// type T0 = NonNullable<string | number | undefined>; // string | number

// type T1 = NonNullable<string[] | null | undefined>; // string[]

// Parameters<T> Constructs a tuple type of the types of the parameters of a function type T.

// declare function f1(arg: { a: number, b: string }): void

// type T0 = Parameters<() => string>; // []

// type T1 = Parameters<(s: string) => void>; // [string]

// type T2 = Parameters<(<T>(arg: T) => T)>; // [unknown]

// type T4 = Parameters<typeof f1>; // [{ a: number, b: string }]

// type T5 = Parameters<any>; // unknown[]

// type T6 = Parameters<never>; // never

// type T7 = Parameters<string>; // Error

// type T8 = Parameters<Function>; // Error

// ConstructorParameters<T> The ConstructorParameters<T> type lets us extract all parameter types of a constructor function type. It produces a tuple type with all the parameter types

// type T0 = ConstructorParameters<ErrorConstructor>; // [(string | undefined)?]

// type T1 = ConstructorParameters<FunctionConstructor>; // string[]

// type T2 = ConstructorParameters<RegExpConstructor>; // [string, (string | undefined)?]

// ReturnType<T> Constructs a type consisting of the return type of function T

// declare function f1(): { a: number, b: string }

// type T0 = ReturnType<() => string>; // string

// type T1 = ReturnType<(s: string) => void>; // void

// type T2 = ReturnType<(<T>() => T)>; // {}

// type T3 = ReturnType<(<T extends U, U extends number[]>() => T)>; // number[]

// type T4 = ReturnType<typeof f1>; // { a: number, b: string }

// type T5 = ReturnType<any>; // any

// type T6 = ReturnType<never>; // any

// type T7 = ReturnType<string>; // Error

// type T8 = ReturnType<Function>; // Error

// Required<T> Constructs a type consisting of all properties of T set to required.

// interface Props {

// a?: number;

// b?: string;

// };

// const obj: Props = { a: 5 }; // OK

// const obj2: Required<Props> = { a: 5 }; // Error: property 'b' missing

// ThisParameterType Extracts the type of the this parameter of a function type, or unknown if the function type has no this parameter.

// Note: This type only works correctly if --strictFunctionTypes is enabled.

// function toHex(this: Number) {

// return this.toString(16);

// }

// function numberToString(n: ThisParameterType<typeof toHex>) {

// return toHex.apply(n);

// }

// console.log(numberToString(4))