**1. Getting Started**

4. Installing & Using TypeScript

**Typescript**-ის დასაინსტალირებლად ვიყენებთ ბრძანებას:

npm install -g typescript

შემდეგ კი ბრძანებით - **tsc  <ჯს ფაილი>**- დააკომპაილებს **TS**-ს **JS**-ში.

**1. Getting Started**

9. The Course Project Setup

ლაივ სერვერისთვის შეგვიძლია **npm**-ის ინიციალიზების მერე, დავაინსტალიროთ სპეციალური პაკეჯი - npm install --save-dev lite-server

შემდეგ კი უნდა გავუშვათ ბრძანება: **lite-server** და ლაივი გაეშვება;

**2. TypeScript Basics & Basic Types**

16. Object Types

**TS**-ში არის პრიმიტიული ტიპები:

**- number**

**- string**

**- boolean**

და ა.შ.

თუ **const**-ში ვინახავთ დეითას, ამ ცვლად ავტომატურად მიენიჭება არა მხოლოდ დეითას ტიპი, არამედ თავად დეითაც, რამდენადაც ესაინმენტი არ ხდება.

თუ **let**-ში ვინახავთ, ამ ცვლადს მიენიჭება მხოლოდ დეითას ტიპი, ანუ სხვა დეითას ესაინმენტისას არ მიიღებს.

**JS**-სსაც აქვს ტიპები, თუმცა ისინი რანთაიმში მოწმდება, **TS**-ის შემთხვევაში კი - დეველოპმენტ თაიმში.

**2. TypeScript Basics & Basic Types**

22. Union Types

ობიექტის ტიპების დეკლარირება რამდენიმენაირად შეიძლება:

1. const person = {
2. name: 'Maxi',
3. age: 30,
4. hobbies: ['Sports', 'Cooking']
5. }
7. const person: object = {
8. name: 'Maxi',
9. age: 30,
10. hobbies: ['Sports', 'Cooking']
11. }
13. const person: {} = {
14. name: 'Maxi',
15. age: 30,
16. hobbies: ['Sports', 'Cooking']
17. }
19. const person: {
20. name: string;
21. age: number; // ასევე შეგვიძლია უბრალოდ რიცხვი დავწეროთ და მის მეტს არ მიიღებს, მაგ - 30
22. hobbies: string[];
23. } = {
24. name: "Maxi",
25. age: 30,
26. hobbies: ["Sports", "Cooking"],
27. };

**2. TypeScript Basics & Basic Types**

22. Union Types

**Tuples**:

1. // Tuples = ტიპი => ფიქსირებული მასივისთვის მაგალითად => (string | number)[] = [1, 'gio]
2. // .push() დაშვებულია ტიუპელებში
4. const personTuple: {
5. name: string;
6. age: number;
7. hobbies: string[];
8. role: [number, string]; // Tuples
9. } = {
10. name: "Maxi",
11. age: 30,
12. hobbies: ["Sports", "Cooking"],
13. role: [1, "gio"],
14. };

**2. TypeScript Basics & Basic Types**

22. Union Types

**Enums:**

1. // Enum = ტიპი => enum {NEW, OLD}
3. enum Role {
4. ADMIN,
5. READ\_ONLY,
6. AUTHOR,
7. }; // ისინი ავტომატურად დაინომრებიან 0++, თუმცა ხელით მანიპულირებაც შეიძლება
9. const personEnum = {
10. name: "Maxi",
11. age: 30,
12. hobbies: ["Sports", "Cooking"],
13. role: Role.ADMIN,
14. };

**2. TypeScript Basics & Basic Types**

22. Union Types

**Union Types: number | string :**

1. // Union Type
2. function combine(input1: number | string, input2: number | string) {
3. let result;
4. if (typeof input1 === "number" && typeof input2 === "number") {
5. result = input1 + input2;
6. }else {
7. result = input1.toString() + input2.toString()
8. }
9. return result;
10. }

**2. TypeScript Basics & Basic Types**

23. Literal Types

ლიტერალ ტიპებში კონკრეტულად ვუთითებთ არა მხოლოდ ტიპს (რომელიც ავტომატურად გენერირდება), არამედ კონკრეტულ სინტაქსსაც, რომელიც უნდა ჰქონდეს ამ ტიპის დეითას, მაგ:

1. function combine(input1: number | string, input2: number | string, resultConversion: 'as-number' | 'as-text') {
2. ...
3. }

**2. TypeScript Basics & Basic Types**

24. Type Aliases / Custom Types

**Type Aliases**  ასევე გამოიყენება ქოსთუმ ტიპების დასადეკლარირებლად:

1. type Combinable = number | string;
3. function combine(input1: Combinable, input2: Combinable, resultConversion: ConversionDescription) {}

ასევე შემდეგნაირადაც შეგვიძლია გამოვიყენოთ:

1. type User = { name: string; age: number };
3. function greet(user: User) {
4. console.log('Hi, I am ' + user.name);
5. }

**2. TypeScript Basics & Basic Types**

26. Function Return Types & "void"

ფუნქციებსაც აქვს რითურნ ტიპები, ასევე თუ არაფერს არ აბრუნებს ფუნქცია, მას დაბრუნების ტიპად void ეწერება, იმისდა მიუხედავად, რომ ფუნქცია, თუ არაფერს არ აბრუნებს - იგი **undefined**-ს აბრუნებს, თუმცა აღნიშნული ტიპი შეიძლება სხვა ქეისში გამოვიყენოთ და არა ფუნქციის დაბუნების ტიპში, რადგან ტექნიკურად ფუნქცია არაა დაშვებული, რომ **undefined**დააბრუნოს:

1. function add(n1: number, n2:number):number {
2. return n1 + n2
3. }
5. function printResult(num: number):void {
6. console.log('Result: ' + num);
7. }
9. let letsomeValue: undefined;

ან უბრალოდ:

1. function printResult1(num: number):undefined {
2. console.log('Result: ' + num);
3. return;
4. }

**2. TypeScript Basics & Basic Types**

27. Functions as Types

ასევე არსებობს ფუნქციები, როგორც ტიპები. სტანდარტულად:

1. let combineValues: Function;

თუმცა ასევე უფრო კონკრეტულებიც შეგვიძლია ვიყოთ ტიპში:

1. let combineValues: (a: number, b:number) => number;
3. function add(n1: number, n2:number):number {
4. return n1 + n2
5. }
6. combineValues = add;

**2. TypeScript Basics & Basic Types**

28. Function Types & Callbacks

ასევე ქოლბაქ ფუნქციასაც ეწერება ტაიპი:

1. function addAndHandle(n1:number, n2: number, cb: (num: number) => void) {
2. const result = n1 + n2;
3. cb(result)
4. }
6. addAndHandle(10, 20, val => {
7. console.log(val)
8. })

**2. TypeScript Basics & Basic Types**

29. The "unknown" Type

ასევე არსებობს ტიპი **unknown** - რომელიც **any**-სგან იმით განსხვავდება, რომ სხვა ტიპის ცვლადთან ვერ გავუკეთებთ ესაინმენტს. მაგ:

1. let userInput:unknown;
3. let userName: string;
5. userInput = 5;
6. userInput = 'Gio';
8. userName = userInput; // ეს დაერორდება

ამის გამოსწორება შეიძლება:

1. let userInput:unknown;
3. let userName: string;
5. userInput = 5;
6. userInput = 'Gio';
8. if(typeof userInput === 'string') {
9. userName = userInput
10. }

**2. TypeScript Basics & Basic Types**

30. The "never" Type

**never** ტიპი გამოიყენება ფუნქციების დაბრუნების ტიპის შემთხვევად, როდესაც ფუნქცია არა **undefined**-ს, არამედ არაფერს არ აბრუნებს, აი, მაგალითად **throw**ქივორდი:

1. function generateError(message: string, code:number): never {
2. throw {
3. message: message, errorCode: code
4. }
5. }
7. const result = generateError('An error occured', 500);

**3. The TypeScript Compiler (and its Configuration)**

34. Using "Watch Mode"

იმისათვის, რომ **tsc**გავუშვათ ლაივ რეჟიმში, უნდა გამოვიყენოთ ბრძანება:

tsc app.ts --watch ან tsc app.ts --w

ასევე თუ გვსურს, რომ რამდენიმე**.ts**ფაილის კომპილირება შევძლოთ და ზოგადად, **TS**-ის კონფიგურირება, შეგვიძლია შემდეგი ბრძანებით **tsconfig.json** ფაილი დავაგენერიროთ:  tsc --init

იგი ასევე ტაიპსკრიპტს ეუბნება, რომ ეს ფოლდერი დამენეჯმენტებული იქნება მის მიერ.

ამის მერე კი მხოლოდ **tsc**ბრძანებით ყველა**.ts** ფაილს დააკომპაილირებს ფოლდერში, ასევე შეგვიძლია ვისარგებლოთ **--watch** დამატებით.

**3. The TypeScript Compiler (and its Configuration)**

36. Including & Excluding Files

**tsconfig.json** - ში შეგვიძლია ბოლოში ჩავამატოთ ახალი ფროფერთიები **json**ფორმატში.

მაგალითად:

1. "exclude": [
2. "analytics.ts"
3. ]

მელიც მიუთითებს, რომ კონკრეტული ფაილი არ დაკომპაილირდეს.

ასევე ვაილდქარდები:

**"\*.dev.ts"**- ყველა ფაილი ამ დაბოლოებით

**"\*\*/\*.dev.ts"**- ნებისმიერ ფოლდერში მსგავსი დაბოლოებით

**"node\_modules"**- უბრალოდ ფოლდერი, რომელიც დეფაულტზე უკვე არ შედის, ასე რომ არაა საჭირო, თუმცა თუ **explude**ქის ვიყენებთ, მაშინ უნდა მივუთითოთ კიდევაც, რომ არ შევიდეს დაკომპაილირების პროცესში.

ასევე არსებობს **includes**, რომელსაც თუ გამოვიყენებთ, მაშინ ყველა ის ფაილი, რომელიც მასსში არ იქნება ნახსენები, არ დაკომპაილირდება:

1. "include": [
2. "app.ts",
3. "analytics.ts"
4. ]

ასევე **files**, სადაც კონკრეტულ ფაილებს ვუთითებთ მხოლოდ, **include**-ისგან განსხვავებით, სადაც ფოლდერებიც მითითებაც შეიძლება:

1. "files": [
2. "app.ts"
3. ]

**3. The TypeScript Compiler (and its Configuration)**

37. Setting a Compilation Target

**tsconfig**-ში უკვე დეკლარირებული ქიების განხილვა:

**"target": "es5"** - მიუთითებს **JS**-ის რომელ ვერსიაში გადაიყვანოს;

**"lib": []** - იგი საკმაოდ გამოყენებადი და კრიტიკულად მნიშვნელოვანი ნაწილია. თუ აღნიშნული დაკომენტარებულია, ანუ არ მუშაობს, **TS**დააყენებს დეფაულტ ბიბლიოთეკებს, ხოლო თუ იგი ჩართულია, **TS**-ს არანაირი დეფაულტ ბიბლიოთეკა აღარ ექნება, შესაბამისად იგი არ იცნობს არც **DOM**-ს და არც **es6**-ის ბიბლიოთეკას. ამის გამოსასწორებლად, საჭირო მოდულებია:

1. "lib": [
2. "DOM",
3. "es6",
4. "DOM.Iterable",
5. "ScriptHost"
6. ],

**"allowJs": true** - იგი ასევე დააკომპაილირებს **JS**ფაილებს;

**"checkJs": true** - იგი არ დააკომპაილირებს, მაგრამ მასში არსებულ სინტაქსს შეამოწმებს და თუ ერორია, გამოიტანს. უკანასკნელი ორი უფრო გამოიყენება, თუ **.ts**ფაილებს არ აკომპაილირებენ და მხოლოდ **.js**-ს ასწორებენ.

გაგრძელდება...

**3. The TypeScript Compiler (and its Configuration)**

39. More Configuration & Compilation Options

**"declaration": true**და **"declarationMap": true** - ესენი უფრო საკუთარი ბიბლიოთეკის შექმნისას გამოიყენება.

**"sourceMap": true** - გამოიყენება დებაგინგისთვის ძირითადად. მისი გააქტიურებისას, დაგენერირდება**.map** გაფართოების ფაილები, რომლებიც ბრაუზერს ესმის თავისთავად და ასევე **Sources**-ში დავინახავთ**.ts** ფაილებსაც.

დიდ პროექტებში **dist**ფოლდერში ძირითადად **.js** ფაილებია, ხოლო **src**ფოლდერში - **.ts** ფაილები.

თუ დეფაულტ სეთთინგებს ვიყენებთ, ეს ასე ვერ იმუშავებს. ამის ორგანიზებაში გვეხმარება ზუსტად:

**"outDir": "./dist"** - სადაც ვუთითებთ, თუ სად უნდა დაგენერირდეს**.js**ფაილები. ასევე, თუ ძირითად ფოლდერში, სადაც **.ts**-ები გვექნება, რაიმე ფოლდერს ჩავამატებთ მაგალითად, **dist**ფოლდერშიც ჩაემატება.

**"rootDir": "./src"** - იგი მკაცრად განსაზღვრავს, თუ რუთი, ანუ .ts ფაილები რომელ ფოლდერში იქნებიან და ასევე მკაცრად აკონტროლებს, რომ dist ფოლდერშიც იგივე სტრუქტურა შენარჩუნდეს.

**3. The TypeScript Compiler (and its Configuration)**

41. rootDir and outDir

**"removeComments": true**- იგი **.ts**-ში ნახსენებ ყველა კომენტარს წაშლის**.js**ვერსიაში.

**"noEmit": true** - იგი არ დააგენერირებს**.ts** ფაილებს. უბრალოდ მათ შემოწმების დონეზე გადახედავს **TS**-ი.

**"downlevelIteration": true** - ეს ლუპების კონტროლისთვის გამოიყენება (ედვანსია).

**"noEmitOnError": true** - იგი გამოიყენება იმისთვის, რომ თუ რაიმე ერორია, ფაილს**.js**-ში აღარ დააკომპაილირებს, არც ერთ ფაილს, რომლებსაც ეს კონფიგურაცია ეხება (ერთშიც რომ იყოს პრობლემა).

**3. The TypeScript Compiler (and its Configuration)**

43. Strict Compilation

**Strict Type-Checking Options:**

აქ **"strict": true** ეს იგივეა, რაც ამ სექციაში ყველას გა-**true**-ება:

**"noImplicitAny": true** - იგი ითხოვს პარამეტრებსა და მნიშვნელობებში ტიპების დეკლარირებას, არა ცვლადებს.

**"strictNullChecks": true** - იგი ყურადღებას აქცევს მოსალოდნელ **null**მნიშვნელობებს და არ გაძლევს საძუალებას მსგავსი ესაინმენტი გამოიყენო, ძირითადად შეეხება **DOM**-ის ელემენტების მონიშვნას.

**"strictFunctionTypes": true** - იგი ფუნქციის ტიპებს შეეხება

**"strictBindCallApply": true** - იგი შეეხება **.bind()** ის ლოგიკის სისწორეს, მაგალითად:

1. function clickHandler(message:string) {
2. console.log('Clicked!' + message)
3. }
4. if(button){
5. button.addEventListener("click", clickHandler.bind(null, "Your're Welcome!"));
6. }

ასე თუ არ იყო, ანუ მარტო **.bind(null)**, რომ იყოს, ერორს ამოაგდებდა.

**"strictPropertyInitialization": true** - კლასებთან მუშაობისას ხდება მნიშვნელოვანი;

**3. The TypeScript Compiler (and its Configuration)**

43. Strict Compilation

**"noImplicitThis": true** - იგი აკონტროლებს **this**ქივორდს რამდენად მართებულად ხმარობ;

**"alwaysStrict": true** - იგი აკონტროლებს, რომ **.js** ფაილები შეიცავენ **'use strict'**-ს თავში;

**3. The TypeScript Compiler (and its Configuration)**

44. Code Quality Options

**Additional Checks**

იგი აკონტროლებს, რომ გამოუყენებელი ცვლადები ან პარამეტრები და ა.შ. არ გქონდეს დარჩენილი კოდში.

*დანარჩენები ედვანსებია და მოგვიანებით იქნება...*

**4. Next-generation JavaScript & TypeScript**

55. How Code Gets Compiled & Wrap Up

**Spread Operator:**

1. //spread operator
2. const hobbies = ['Sports', 'Cooking']
3. const activeHobbies = ['Hiking']
5. activeHobbies.push(...hobbies)
6. hobbies.push('gogo')
8. const person = {
9. name: 'Gio',
10. age: 30
11. }
13. const copiedPerson = {
14. profession: 'Doctor of Computer',
15. person: {...person}
16. }

**4. Next-generation JavaScript & TypeScript**

55. How Code Gets Compiled & Wrap Up

**Rest Parameters:**

1. // rest parameters
2. const addSmth = (...numbers: number[]):number => {
3. return  numbers.reduce((a,b) => a+b)
4. }
6. const addedNumbers = addSmth(5,10,2,3.7)
7. console.log(addedNumbers)

**4. Next-generation JavaScript & TypeScript**

55. How Code Gets Compiled & Wrap Up

**Object Destructuring - Arrays and Objs:**

1. // object destructuring
2. //array
3. const newHobbies = ['Sports', 'Cooking', 'Pooping', 'Hating being here', 'and etc.']
4. const [hobby1, hobby2, ...remainingHobbies] = newHobbies
5. console.log(hobby1, hobby2, remainingHobbies)
7. //obj
8. const newPerson = {
9. firstName: 'Gio',
10. age: 30,
11. wish: 'Get me out of here',
12. dream: 'flying...'
13. }
14. const { firstName: userName, dream, wish } = newPerson
16. console.log(userName, dream, wish)

**5. Classes & Interfaces**

61. Constructor Functions & The "this" Keyword

ცლასების ნეიმინგ კონვენციაა - **დიდი ასოთი დაწყება**

**constructor**გაეშვება მაშინ, როცა ობიექტი შეიქმნება **new**- ქივორდით, ანუ ახალი ინსტანსი. იგი ინიციალიზებაში იღებს მონაწილეობას - ფროფერთიების და მნიშვნელობების

1. class Department {
2. depName:string;
3. constructor(n:string) {
4. this.depName = n
5. } // constructor გაეშვება მაშინ, როცა ობიექტი შეიქმნება new - ქივორდით, ანუ ახალი ინსტანსი. იგი ინიციალიზებაში იღებს მონაწილეობას - ფროფერთიების და მნიშვნელობების.
7. describe() {
8. console.log('Department: ' + this.depName)
9. }
11. solveDescribe(this:Department) {
12. console.log('Department: ' + this.depName)
13. }
14. // ამ შემთხვევაში Type-Safety-ს განვსაზღვრავთ, რათა სხვა ობიექტის მიერ არ მოხდეს ამ მეთოდის გამოძახება. this არგუმენტად არ ნიშნავს რომ არგუმენტს მოითხოვს მეთოდი.
16. }

*გაგრძელდება...*

**5. Classes & Interfaces**

61. Constructor Functions & The "this" Keyword

ასევე მნიშვნელოვანია (წინა კლასის გაგრძელებით):

1. const accounting = new Department('Accounting') // შეიქმნება ობიექტი class-ის ბლუპრინტზე დაყრდნობით
3. accounting.describe()
5. // JS-ში კომპილირების შემდეგ, name ცალკე ფილდად არ იარსებებს, რადგან ჯერ es6 მას მხარს არ უჭერს
7. const acccountingCopy = { describe: accounting.describe }
8. acccountingCopy.describe() // undefined
10. // this - ქივორდი ამ უკანასკნელ შემთხვევაში არ იმუშავებს, რადგან იგი მიმართავს იმ ობიექტს, რომელიც პასუხისმგებელია მის გამოძახებაზე. ვინაიდან ეს უკანასკნელი არ ფლობს depName ფროფერთის, იგი undefined იქნება
12. const accountingCopy1 = {depName: 'Solving prop!', solveDescribe: accounting.solveDescribe, describe: accounting.describe}
14. accountingCopy1.solveDescribe()
16. // მხოლოდ ამ უკანასკნელ შემთხვევაში შევძლებდით გამოსწორებას, ანუ თუ მივამსგავსებდით იგივე ობიექტს, რომლიდანაც მოაქვს მეთოდი.

**5. Classes & Interfaces**

62. "private" and "public" Access Modifiers

წინა კლასში ასევე შეგვიძლია:

1. class Department {
2. public depName:string; // დეფაულტზეა
3. private employees: string[] = []; // პირდაპირ გარედან ვერ მიწვდებიან ობიექტში ამ ფროფერთის. იგი ასევე მეთოდებთანაც შეგიძლია გამოიყენო და ეგ მეთოდები მხოლოდ შიგნით იქნება ხელმისაწვდომი.
4. ...
5. }
7. const accounting = new Department('Accounting') // შეიქმნება ობიექტი class-ის ბლუპრინტზე დაყრდნობით
9. accounting.addEmployee('Max')
10. accounting.addEmployee('Manu')
12. // accounting.employees.push('Anna') // ეს არაა სასურველი
14. accounting.printEmployeeInformation()

**5. Classes & Interfaces**

63. Shorthand Initialization

წინა კლასის მაგალითზე, ასევე შეგვიძლაი ფროფერთიების შექმნა კონსტრუქტორიდან, თუ ფროფერთის წინ გავუწერთ აქსეს მოდიფაიერს (ანუ საჯაროა თუ არა):

1. constructor(private id:string, public depName:string) {
2. this.depName = depName;
3. this.id = id
4. }

**5. Classes & Interfaces**

64. "readonly" Properties

**readonly**- იგი ნიშნავს, რომ ამ ფროფერთის ინიციალიზების მერე, ანუ რაც კონსტრუქტორში მოხდება, მის მერე ვეღარ შევცვლით:

1. constructor(private readonly id:string, public depName:string) {
2. this.depName = depName;
3. this.id = id
5. /// კონსტრუქტორშივე თუ ვუწერთ აქსეს მოდიფაიერს, იგი ავტომატურად შეუქმნის ობიექტს ფროფერთიებს
7. // readonly - იგი ნიშნავს, რომ ამ ფროფერთის ინიციალიზების მერე, ანუ რაც კონსტრუქტორში მოხდება, მის მერე ვეღარ შევცვლით
8. }

**5. Classes & Interfaces**

65. Inheritance

**Inheritance**- მემკვიდრეობითობა - შეიძლება მხოლოდ ერთ კლასზე გავრცელდეს, ანუ ორ კლასს ვერ დავაექსტენდებთ - **extends**

1. class Department {
2. private employees: string[] = [];
4. constructor(private readonly id:string, public name:string) {}
6. describe(this: Department) {
7. console.log(`Department: (${this.id}): ${this.name}`)
8. }
10. addEmployee(employee:string) {
11. this.employees.push(employee)
12. }
14. printEmployeeInformation() {
15. console.log(this.employees.length);
16. console.log(this.employees)
17. }
18. }
19. class ItDepartment extends Department{}

იგი დაექსტენდების მერე, ყველაფერს იღებს ზედა კლასისგან, მათ შორის კონსტრუქტორსაც:

1. const accounting = new ItDepartment('d1','Accounting')

**5. Classes & Interfaces**

65. Inheritance

**super()**გამოიყენება **base**, ანუ მეკვიდრე კლასის კონსტრუქტორში. იგი იძახებს მშობელი კლასის კონსტრუქტორს და აწვდის არგუმენტებს. იგი ასევე კონსტრუქტორის სახეს უცვლის ახალ კლასს. ყველაფერი, მათ შორის **this**ქივორდი გამოძახებულ უნდა იქნას **super()**-ის შემდეგ.

შემდეგ კი ჩვეულებრივ, ამ მემკვიდრე კლასის კონსტრუქტორს შეგვიძლია ჩავაწოდოთ ახალი არგუმენტები, მათ შორის შევაქმნევინოთ ფროფერთიები, რომლებიც მხოლოდ ამ კლასს მიეკუთვნებიან.

***წინას გაგრძელება:***

1. class ItDepartment extends Department{
2. admins:string[]
3. constructor(id: string, admins: string[]) {
4. super(id, 'IT');
5. this.admins = admins
6. }
7. }

**5. Classes & Interfaces**

66. Overriding Properties & The "protected" Modifier

ასევე საყურადღებოა, რომ **private**მეთოდი კონკრეტულ ფროფერთის ან მეთოდს ხელმისაწვდომს ხდის მხოლო იმ კლასში, რომელშიცაა დეკლარირებული და **extends**ქივორდით სხვა კლასში ინჰერიტანსით იგი არ მიჰყება, ანუ იმ კლასებში რომლებიც მას აექსთენდებენ, ვერ გამოვიყენებთ (არა მხოლოდ გარეთ). ამის გამოსასწორებლად, შეგვიძლია გამოვიყენოთ **protected**ქივორდი, რომელიც დაიცავს ფროფერთის ან მეთოდს გარედან შეღწევისგან და ასევე მას ხელმისაწვდომს გახდის ინჰერიტანსშიც.

***მაგალითი წინადან:***

1. class Department {
2. protected employees: string[] = [];
3. }
5. class AccountingDepartment extends Department {
6. constructor(id: string, private reports: string[]){
7. super(id, 'Accounting');
8. }
9. addEmployee(name: string) {
10. if(name === 'Gio') return;
11. this.employees.push(name)
12. }
13. }

**5. Classes & Interfaces**

67. Getters & Setters

კლასებში ასევე შეგვიძლია **getter**-ები და **setter**-ები გამოვიყენოთ.

თუ მათით ვისარგებლებთ: **get**-ის შემთხვევაში, არგუმენტის ჩაწოდება აღარ გვჭირდება, შესაბამისად როცა ვიძახებთ, ფროფერთის სახით ვწერთ

**set**-ის შემთხვევაში კი, არგუმენტს ვაწვდით და გამოძახებისას ისევ ფროფერთის სახით და რა მნიშვნელობაც უნდა ჩავაწოდოთ, უბრალოდ მას ვუტოლებთ

1. class AccountingDepartment extends Department {
2. private lastReport: string;
4. get getMostRecentReport() {
5. if (this.lastReport) return this.lastReport;
6. throw new Error("No report found.");
7. }
9. set setMostRecentReport(value: string) {
10. if (!value) throw new Error("Please pass in a valid value!");
11. this.lastReport = value
12. }
14. constructor(id: string, private reports: string[]) {
15. super(id, "Accounting");
16. this.lastReport = reports[0];
17. }
18. }
19. // setter call:
20. accounting.setMostRecentReport= "Year End Report"
21. // getter call:
22. accounting.getMostRecentReport

**5. Classes & Interfaces**

68. Static Methods & Properties

**static**მეთოდები ან ფროფერთიები ხელმისაწვდომია მხოლოდ ძირითად კლასში, ისინი ასევე არც ახალ ინსტანსებზე არ ვრცელდებიან. ასევე მნიშვნელოვანია ვიცოდეთ, რომ არასტატიკური მეთოდებიდან და ფროფერთიებიდან ვერ მივწვდებით სტატიკურ მეთოდებს და ფროფერთიებს.

მათ გამოსაყენებლად ერთადერთი გზაა კლასის სახელით მივწდეთ. მაგ.: **Department.fiscalYear**

***წინა მაგალითიდან:***

1. class Department {
2. ...
3. static fiscalYear = 2020;
5. constructor(private readonly id: string, public name: string) {}
7. static createEmployee(name: string) {
8. return { name: name };
9. }
10. ...
11. }

14. console.log(Department.createEmployee("Gio")) // {name:'gio'}
15. console.log(Department.fiscalYear) // 2020

**5. Classes & Interfaces**

69. Abstract Classes

ასევე შესაძლებელია მეთოდების გადაწერა. მაგ:

***წინას გაგრძელება:***

1. class Department {
2. constructor(protected readonly id: string, public name: string) {}
4. describe(this: Department) {
5. console.log(`Department: (${this.id}): ${this.name}`);
6. }
7. }
9. class AccountingDepartment extends Department {
11. constructor(id: string, private reports: string[]) {
12. super(id, "Accounting");
13. this.lastReport = reports[0];
14. }
16. describe() {
17. console.log(`Accounting Department - ID: ${this.id}`)
18. }
20. }
22. const accounting = new AccountingDepartment("d2", []);
23. accounting.describe() // Accounting Department - ID: d2

**5. Classes & Interfaces**

69. Abstract Classes

გარდა იმისა, რომ მეთოდებისა და ფროფერთიების გადაწერა შეგვიძლია, ასევე შეგვიძლია ცენტრალურ კლასში შევქმნათ მეთოდი, რომელსაც თავიდანვე ვაიძულებთ, რომ იყოს აბსტრაქტული, ანუ მისი შთამომავალი კლასები იძულებულები იქნებიან ამ მეთოდის იმპლემენტირება მოახდინონ, ანუ შექმნან მათეული ვერსია. ამისთვის მეთოდს წინ ვუწერთ abstract და მას არ ექნება ფუნქციის ბლოკი, უბრალოდ ტიპი, რასაც აბრუნებს: **abstract describe(this: Department): void**; თუმცა ამის ასამუშავებლად, ძირითადი კლასიც ამ ქივორდით უნდა დაიწყოს:**abstract class Department {}**

აბსტრაქტული ასევე ფროფერთიც შეიძლება იყოს;

ასევე მნიშვნელოვანია, რომ ვიცოდეთ: აბსტრაქტული კლასის ინსტანსის შექმნა შეუძლებელია. მხოლოდ იმ კლასების ინსტანსები იქმნება, რომლებიც მას აექსტენდებენ

*წინიდან:*

1. abstract class Department {
2. constructor(protected readonly id: string, public name: string) {}
3. abstract describe(this: Department): void;
4. }
6. class ItDepartment extends Department {
7. constructor(id: string) {}
8. describe() {
9. console.log(...)
10. }
11. }
13. const it = new ItDepartment("d1", ["Gio"]);
14. it.describe()

**5. Classes & Interfaces**

70. Singletons & Private Constructors

**Singleton OOP**-ში ნიშნავს - იმაში დარწმუნებულობას, რომ ერთი კლასის მხოლოდ ერთი ინსტანსი გაქვს, ანუ მხოლოდ ერთი ობიექტი გაქვს კლასის ბლუპრინტზე დაყრდნობით.

ამის მისაღწევად **constructor()**-თან ვიყენებთ **private**ქივორდს რომელიც მას გლობალურად მიუწვდომელს ხდის, ხოლო ამ კლასში შიგნითვე **static**მეთოდებისა და ფროფერთიების წყალობით, ვქმნით ისეთ მეთოდს, რომელიც მხოლოდ ერთ ინსტანსს შექმნის ამ კლასის. გამოძახებისას კი, ამ კონკრეტულ მეთოდს გამოვიძახებთ, რათა იმუშავოს.

მაგალითი:

1. abstract class Department {...}
3. class AccountingDepartment extends Department {
4. private lastReport: string;
5. private static instance: AccountingDepartment;
7. private constructor(id: string, private reports: string[]) {
8. super(id, "Accounting");
9. this.lastReport = reports[0];
10. }
12. static getInstance() {
13. if(AccountingDepartment.instance){
14. return this.instance
15. }
16. this.instance = new AccountingDepartment('d2', [])
17. return this.instance;
18. }
20. }
22. const accounting = AccountingDepartment.getInstance();

**5. Classes & Interfaces**

72. A First Interface

**Interface**- განსაზღვრავს როგორ უნდა გამოიყურებოდეს ობიექტი, ანუ სტრუქტურა.

1. interface Person {
2. name: string;
3. age: number;
5. greet(phase: string): void;
6. }
8. let user1: Person;
10. user1 = {
11. name: "Gio",
12. age: 26,
14. greet(phase: string) {
15. console.log(phase);
16. },
17. };
19. user1.greet("xaloo");

**5. Classes & Interfaces**

73. Using Interfaces with Classes

**interface**-ებსა და **type**-ებს შორის ის განსხვავებებია: **interface**მხოლოდ ობიექტების ტიპებისთვის გამოიყენება, ხოლო **type**სხვა ტიპებისთვისაც, ასევე **interface**ხაზს უფრო ობიექტს უსვამს, ამიტომ ობიექტის ტიპის დასადეკლარირებლად ხშირად გამოიყენება

**interface**ასევე შეუძლია დაიმპლემენტირდეს კლასის მიერ:

1. interface Greetable {
2. name: string;
4. greet(phase: string): void;
5. }
7. class Person implements Greetable {
8. name: string;
9. age = 30;
10. constructor(n: string) {
11. this.name = n;
12. }
13. greet(phase: string) {
14. console.log(phase);
15. }
16. }
18. const user1: Greetable = new Person('Giorgi')
19. user1.greet('Hello')

ამ მხრივ კი ამ კლასს შეუძლია უფრო მეტი რამ ჰქონდეს, თუმცა რასაც დაიმპლემენტებს, მისი მოთხოვნები უნდა დააკმაყოფილოს რომ არ დაერორდეს

ასევე ინტერფეისებსა და აბსტრაქტ კლასებს შორის ის განსხვავებაა, რომ კლასებს იმპლემენტირებული აქვთ მეთოდები, რომლებიც ახალ ინსტანსში გადადიან მათი ფუნქციონალით, ინტერფეისი კი უბრალოდ სტრუქტურის მოდელს ქმნის.

**5. Classes & Interfaces**

76. Extending Interfaces

**readonly**ასევე შეიძლება მიემაგროს ფროფერთის ან მეთოდს ინტერფეისში, რაც იგულსიხმებს, რომ მას შემდეგ, რაც ობიექტის ინიციალიზება მოხდება, აღარ შეიცვალოს. (ასევე **type**-შიც შეგვიძლია გამოვიყენოთ **readonly**).

1. interface Greetable {
2. readonly name: string;
4. greet(phase: string): void;
5. }

**5. Classes & Interfaces**

76. Extending Interfaces

**interface**ასევე კლასის მსგავსად აქვს **inheritance**და იგივე პრინციპით მუშაობს **extends**ქივორდის გამოყენების შემდეგ.

**implements & extends** ორივე შემთხვევაში მძიმით შეგვიძლია გამოვყოთ თუ ერთზე მეტს აიმპლემენტებს ან აექსთენდებს (კლასებში **extends**მხოლოდ ერთ კლასზეა შესაძლებელი, ინტერფეისში - რამდენზეც გვსურს)

1. interface Named {
2. readonly name:string
3. }
5. interface Greetable extends Named {
6. greet(phase: string): void;
7. }

**5. Classes & Interfaces**

77. Interfaces as Function Types

**interface**ასევე შეიძლება გამოიყენო ფუნქციის ტიპის დასადეკლარირებლად, რომელიც ფუნქციის ტიპის ალტერნატივაა

**type:**

1. type AddFn = (a: number, b: number) => number;
2. let add: AddFn;
3. add = (n1: number, n2: number):number => n1+n2;

**interface:**

1. interface AddFn1 {
2. (a: number, b:number):number;
3. }
4. let add1:AddFn1;
5. add1 = (n1: number, n2:number):number => n1 + n2

**5. Classes & Interfaces**

78. Optional Parameters & Properties

**?** - კითხვის ნიშნის დამატებით შეგვიძლია ინტერფეისში კონკრეტული მეთოდი ან ფროფერთი ოფშენალი გავხადოთ. შემდეგ **TS**-მა რომ არ გაატრაკოს, ! - ნიშნით უნდა მივანიშნო, რომ დარწმუნებული ვარ იგი დააბრუნებს მნიშვნელობას, ან იარსებებს, სხვა მხრივ იგი ჩათვლის, რომ ოფშენალ მეთოდი ან ფროფერთი შეიძლება არ იყოს და დააერორებს, ალტერნატივაა - ქონდიშენალ ლოგიკით შევამოწმოთ და **TS**მიხვდება, რომ უკვე ვიცით, რომ შეიძლება არაფერი არ დააბრუნოს ან არ იარსებოს, ან უბრალოდ ინიციალიზება უნდა მოახდინო კონკრეტული ფროფერთის მაგალითად.

იგი ასევე შეიძლება იგივენაირად მოხდეს კლასშიც

ასევე კლასის კონსტრუქტორშიც შეიძლება პარამეტრები ოფშენალი იყოს ისევ კითხვის ნიშნის გამოყენებით

**interface:**

1. interface Greetable {
2. greet?(phase: string): void;
3. outputName?: string;
4. }

class:

1. class Person implements Greetable {
2. name?: string;
3. age = 30;
4. constructor(n?: string) {
5. this.name = n;
6. }
7. greet(phase: string) {
8. console.log(phase);
9. }
10. }

**6. Advanced Types**

83. Intersection Types

**Date**ასევე ტიპს წარმოადგენს, რომელიც დაფუძნებულია **Date**-ის ობიექტზე, რომელიც **JS**-შია ჩაშენებული

**intersection**ტიპები ის ტიპებია, რომლებიც სხვა ტიპებს აერთიანებენ. იგი შეიძლება იყოს როგორც **type**, ასევე **interface**:

1. type Admin = {
2. name: string;
3. privileges: string[]
4. }
6. type Employee = {
7. name: string;
8. startDate: Date;
9. }
11. type ElevatedEmployee = Admin & Employee
13. const e1: ElevatedEmployee = {
14. name: 'Gio',
15. privileges: ['create-server'],
16. startDate: new Date()
17. }
18. //\*\*\*
19. interface ElevatedEmployee1 extends Employee, Admin {} //ესეც შეიძლება

**6. Advanced Types**

84. More on Type Guards

*ტიპები ზოგი აქ არ ჩანს...*

**typeguard**-ს წარმოადგენს მაგალითად ქონდიშენალ შემოწმება (**typeof**):

1. function add(a: Combinable, b: Combinable): Combinable {
2. if (typeof a === "string" || typeof b === "string") {
3. return a.toString() + b.toString();
4. }
5. return a + b;
6. }

ასევე შეგვიძლია **JS**-ში არსებული სინტაქსით, როგორც ზევით, ობიექტის ფროფერთიების შემოწმება, რომელსაც **TS**-იც ხვდება, როგორც ზევით.

ამისთვის გამოვიყენებთ **in**სინტაქსს: **if('smth' in obj)**, რომელიც შეამოწმებს, კონკრეტული ფროფერთი არსებობს თუ არა ობიექტში:

1. type UnknownEmployee = Employee | Admin;
2. function printEmoloyeeInformation(emp: UnknownEmployee) {
3. console.log("Name: " + emp.name);
4. if ("privileges" in emp) {
5. console.log("Privileges: " + emp.privileges);
6. }
7. if ("startDate" in emp) {
8. console.log("Start Date: " + emp.startDate);
9. }
10. }

*გაგრძელდება...*

**6. Advanced Types**

84. More on Type Guards

მსგავსი გზაა, ოღონდ უფრო ინსტანსებზე ვრცელდება, **instanceof**, რომელიც უბრალოდ ამოწმებს, კონკრეტული ობიექტი არის თუ არა კონკრეტული კლასის ინსტანსი და თუ არის, **TS**ამას აღმოაჩენს და მასზე გაავრცელებს მის მეთოდებსაც. მაგალითი:

1. class Car {
2. drive() {
3. console.log("Driving...");
4. }
5. }
7. class Truck {
8. drive() {
9. console.log("Driving a truck...");
10. }
11. loadCargo(amount: number) {
12. console.log("Loading cargo ..." + amount);
13. }
14. }
16. type Vehicle = Car | Truck;
18. const v1 = new Car();
19. const v2 = new Truck();
21. function useVehicle(vehicle: Vehicle) {
22. vehicle.drive();
24. //ასეც შეიძლებოდა:
26. // if('loadCargo' in vehicle) {
27. //     vehicle.loadCargo(100)
28. // }
30. //თუმცა, instanceof გამოვიყენეთ:
32. if (vehicle instanceof Truck) {
33. vehicle.loadCargo(100);
34. }
35. }

**6. Advanced Types**

85. Discriminated Unions

**discriminated unions** - არიან უბრალოდ ინტერფეისის დამატებითი ფროფერთიები, ჩვეულებრივ - **type**- რომელიც ინახავს ლითერალ ტიპს, ანუ მაგალითად სტრინგს, რომელშიც წერია რაღაც. შემდეგ კი, **switch**-ით კონკტროლდება, რომელი **type**გამართლდა, ანუ რომელი **type**-ის ობიექტი ჩაეწოდა:

1. interface Bird {
2. type: 'bird';
3. flyingSpeed: number;
4. }
6. interface Horse {
7. type: 'horse';
8. runningSpeed: number;
9. }
11. type Animal = Bird | Horse;
13. function moveAnimal(animal: Animal) {
14. let speed;
15. switch(animal.type) {
16. case 'bird':
17. speed = animal.flyingSpeed;
18. break;
19. case 'horse':
20. speed = animal.runningSpeed;
21. }
22. console.log('Moving with speed: ' + speed)
23. }
25. moveAnimal({type: 'bird', flyingSpeed: 5})

**6. Advanced Types**

86. Type Casting

**Type Casting** - გამოიყენება, როცა **TS**ვერ შეძლებს ტიპის ამოცნობას, თუმცა ჩვენ უკვე ვიცით რა ტიპის იქნება.

მისი ხშირად გამოყენება ხდება **DOM**-თან მუშაობისას. იგი წინ ეწერება კონკრეტულ ელემენტს **<>** - სინტაქსით:

1. const userInputElement = <HTMLInputElement>document.getElementById('user-input');
3. userInputElement.value = 'Hi there! <>';

ასევე არსებობსმ ისი ალტერნატივაც **as**ქივორდით და შემდეგ ტიპის გაწერით (იგი უფრო რექტში გამოიყენება, რადგან ზემოთ ნახსენები სინტაქსის სხვაგამოყენებაც აქვთ):

1. const userInputElement1 = document.getElementById('used-input') as HTMLInputElement;
3. userInputElement1.value = 'Hi here! as';

*გაგრძელდება...*

**6. Advanced Types**

86. Type Casting

ასევე **!** - ამით **TS**-ს ვეუბნებით, რომ კონკრეტული ელემენტი არ დააბრუნებს **null**-ს, გამოიყენება **nullish**ელემენტებთან, მაგ.:

1. const userInputElementa = <HTMLInputElement>document.getElementById('user-input')!;

მისი ალტერნატივაა**()**-ის გამოყენება, რომელიც JS-ის ოპერატორების შესრულების პრინციპის (**PEMDAS**მათ შორის) მიხედვით, პირველი შეასრულოს კონკრეტული მოქმედება:

1. const userInputElementa = <HTMLInputElement>document.getElementById('user-input');
2. if(userInputElementa) {
3. (userInputElementa as HTMLInputElement).value = 'How are you?'
4. }

**6. Advanced Types**

87. Index Properties

**Index types**

როდესაც წინასწარ არ ვიცით, რა სახის ფროფერთიები შევა ობიექტში, თუმცა ვიცით, რომ იგი კონკრეტული ტიპის იქნება (გარდა **boolean**-ისა), შეგვიძლია**[]**-ის სინტაქსით განვსაზღვროთ, რომ მაგალითად, იგი იქნება სტრინგი და მისი მნიშვნელობაც იქნება სტრინგი:

1. interface ErrorContainer {
2. [prop: string]: string;
3. }

აღნიშნული სხვა ტიპის ფროფერთის აღარ დაუშვებს კონკრეტულ ინტერფეისში ან იმ ობიექტში, რომლის ტიპიზაციასაც ეს ინტერფეისი ახდენს:

1. const errorBag: ErrorContainer = {
2. email: 'Not a valid email',
3. username: 'Must start with a capital character!'
4. // ფროფერთი რიცხვიც შეიძლება იყოს, რადგან იგი საბოლოოდ სტრინგად გადაიქცევა
5. }

**6. Advanced Types**

88. Function Overloads

**Function overloads** - იგი წარმოადგენს ფუნქციის მიერ დასაბრუნებელი დეითას დედიქშენს, ანუ ზუსტად განსაზღვრას, თუ ფუნქცია რომელ კონკრეტულ შემთხვევაში რას დააბრუნებს - ეს, რა თქმა უნდა, იმ შემთხვევაში, თუ ფუნქცია ფლობს უნიონ ტიპს.

ეს ხდება იგივე სახელწოდების ფუნქციის ზევით დაწერით და კონკრეტული ქეისის განსაზღვრით, კოდის ბლოკის გარეშე.

ასევე შეიძლება ოფშენალ პარამეტრებით, პარამეტრის შემცირებაც კი

მაგალითი:

1. function daamate(n:number):number; // ოფშენალად აქცია მეორე პარამეტრი
2. function daamate(a: number, b?: number): number;
3. function daamate(a: string, b?: string): string;
4. function daamate(a: Combinable, b?: Combinable): Combinable {
5. if (typeof a === "string" || typeof b === "string") {
6. return a.toString() + b!.toString();
7. }
8. return a + b!;
9. }

**6. Advanced Types**

89. Optional Chaining

**Optional Chaining** - იგი გამოიყენება მაშინ, თუ მაგალითად დეითაბეიზიდან ვიღებთ ინფოს და არ ვიცით ეს ინფო როგორი ტიპის იქნება

1. const fetchedUserData = {
2. id: 'u1',
3. name: 'Max',
4. // job რომ არ არსებობდეს
5. job: {
6. title: 'CEO',
7. description: 'My own company'
8. }
9. }

**JS**-ში ასე შევამოწმებდით:

1. console.log(fetchedUserData.job && fetchedUserData.job.title)

**TS**-ში ასე იქნებოდა - **Optional Chaining**:

1. console.log(fetchedUserData?.job?.title)

**6. Advanced Types**

90. Nullish Coalescing

**Nullish Coalescing** - იგი მყიფედ, მაგრამ მაინც, ოფშენალ ჩეინინგის ალტერნატივადაც განიხილება

1. const userInput = null;

**JS**-ში (თუმცა პრობლემურია, ზოგ შემთხვევაში, თუ ნულიშს მნიშნველობა აქვს):

1. const storedData = userInput || 'DEFAULT';

**TS**-ში - **??** გამოიყენება, რომელიც ცარიელ სტრინგს არ ჩათვლის ფოლსად, თუმცა **undefined**-ს ჩათვლის

1. const storedDataTS = userInput ?? 'DEFAULT';