**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. }