Understanding TypeScript - 2020 Edition

# Section 1: Getting Started

## 01. Welcome to the Course!

## 02. Why Is TypeScript & Why Should You Use It?

## 3. Installing & Using TypeScript

### ! as HTMLInputElement;

const input1 = document.getElementById("num1")! as HTMLInputElement;

const input2 = document.getElementById("num2")! as HTMLInputElement;

inseamna ca nu este null si este un HTML input element!

button.addEventListener("click", function() {

  console.log(add(+input1.value, +input2.value));

  const result = add(+input1.value, +input2.value)

  document.getElementById('result').innerHTML = result;

});

### +input1.value

**+ transforma string to number**

## 4. TypeScript Advantages – Overview

## 5. Course Outline

## 6. How To Get The Most Out Of The Course

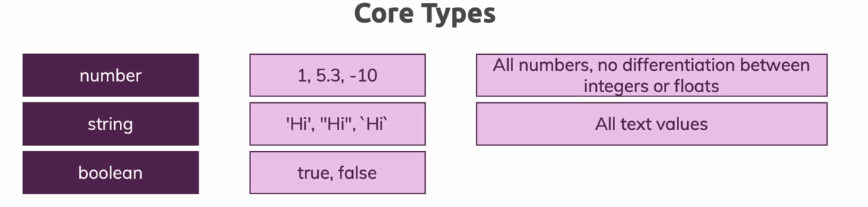
## 7. Setting Up A Code Editor / IDE

## 8. The Course Project Setup

# Section 2: TypeScript Basics & Basic Types

## 9. Module Introduction

## 10. Using Types



function add(n1, n2) {

  return n1 + n2

}

const number1 = 5;

const number2 = 2.8;

const result = add(number1, number2)

console.log(result); //7.8

const number1 = '5';

console.log(result); //52.8

**Error**

**app.ts:8:20 - error TS2345: Argument of type '"5"' is not assignable to parameter of type 'number'.**

**8 const result = add(number1, number2)**

## 11. TypeScript Types vs JavaScript Types

**Ptr. js**

function add(n1:number, n2:number) {

if(typeof n1 !== 'number' || typeof n2 !== 'number') {

  throw new Error('Incorrect input!');

}

  return n1 + n2

}

const number1 = '5';

const number2 = 2.8;

const result = add(number1, number2)

console.log(result); //7.8

app.js:3 Uncaught Error: Incorrect input!

at add (app.js:3)

at app.js:9

add @ app.js:3

(anonymous) @ app.js:9

## 12. Important: Type Casing

### string

### number

In TypeScript, you work with types like string or number.

**Important**: It is string and number (etc.), **NOT** String, Number etc.

**The core primitive types in TypeScript are all lowercase!**

## 13. Working with Numbers, Strings & Booleans

function add(n1:number, n2:number, showResult: boolean, phrase:string) {

  const result = n1 + n2

  if(showResult) {

    console.log(phrase + result)

  } else {

    return result

  }

}

const number1 = 5;

const number2 = 2.8;

const printResult = true;

const resultPhrase = 'Result is: '

const result = add(number1, number2, printResult, resultPhrase )

## 14. Type Assignment & Type Inference

### Type Inference

Pentru aceste variabile nu trebuie as mai declaram type-ul! Deoarece ar fi redundant!

const number1 = 5;

const number2 = 2.8;

const printResult = true;

const resultPhrase = 'Result is: '

insa…

Daca avem acest caz:

//caz

let number1;

number1 = '5';

trebuie sa declaram tipul variabile number1

**correct!!**

//caz

let number1:number;

number1 = 5;

## Quiz 1: Understanding Types

**Why are "Types" useful and offer an advantage compare to vanilla JavaScript?**

Good job!

That's correct! Types allow you to detect if data of a wrong type is used or pass around, hence you can avoid unexpected runtime errors.

**Will the following code throw a compilation error?**

1. **let userName: string;**
2. **userName = 'Maximilian';**
3. **userName = false;**

yes, assigning a boolean to a variable which was assigned a "string" types is not allowed and will yield a compilation error.

**Does this code rely on type inference?**

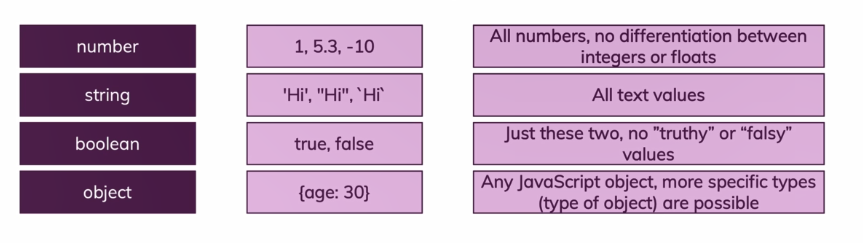
1. **const age: number = 29;**

 TypeScript would be able to infer the type (and hence you should omit ":number") but here, we actually also have an explicit type assignment.

**What's the difference between JavaScript types (e.g. typeof 'Max' => 'string') and TypeScript types (e.g. const name: string = '...')?**

 JS has no compilation step but at runtime, you can check for certain types (e.g. in if conditions). TS on the other hand allows you to catch certain errors during development since it checks types during compilation as well.

## 15. Object Types



const person:{

  name:string;

  age:number;

} = {

  name: 'Max',

  age: 30

};

console.log(person.name)

## 16. Nested Objects & Types

Of course object types can also be created for**nested objects**.

Let's say you have this JavaScript **object**:

const product = {

  id: 'abc1',

  price: 12.99,

  tags: ['great-offer', 'hot-and-new'],

  details: {

    title: 'Red Carpet',

    description: 'A great carpet - almost brand-new!'

  }

}

This would be the **type** of such an object:

{

  id: string;

  price: number;

  tags: string[],

  details: {

    title: string;

    description: string;

  }

}

So you have an object type in an object type so to say.

Pentru a functiona asta trebie pusa in **interface!**

### Interface

interface person {

  name:string;

  age:number;

}

const person:person = {

  name: 'Max',

  age: 30

};

console.log(person.name)

interface product {

  id: string;

  price: number;

  tags: string[],

  details: {

    title: string;

    description: string;

  }

}

const product = {

  id: 'abc1',

  price: 12.99,

  tags: ['great-offer', 'hot-and-new'],

  details: {

    title: 'Red Carpet',

    description: 'A great carpet - almost brand-new!'

  }

}

console.log(product.id)

## 17. Arrays Types



//17. Arrays |Types

const member = {

  name: 'Max',

  age: 30,

  hobbies: ['Sports', 'Cooking']

};

let favoriteActivities: string[];

favoriteActivities = ['sport']

**Error**

favoriteActivities = ['sport', 5]

for (const hobby of member.hobbies) {

  console.log('hobby:', hobby);

}



## 18. Working with Tuples



//18.Tuples

member.role.push('admin');

member.role[1] = 10;

for (const r of member.role) {

  console.log('role:', r);

}

const member: {

  name: string,

  age: number,

  hobbies: string[],

  role:[number, string]

} = {

  name: 'Max',

  age: 30,

  hobbies: ['Sports', 'Cooking'],

  //tuples

  role: [2, 'author']

};

### Tuples

Ptr tuples folosim

  role:[number, string]

/18.Tuples

//an array with 2 values

//member.role.push('admin');

 member.role[0] = 10;

for (const r of member.role) {

  console.log('role:', r);

}

console.log(member.role)

## 19. Working with Enums

### enums

//19.Enums

const ADMIN = 0;

const READ\_ONLY = 1;

const AUTHOR  = 2;

const person2 = {

  name: 'Max',

  age: 30,

  hobbies: ['Sports', 'Cooking'],

  role:ADMIN

};

if(person2.role === ADMIN) {

  console.log('is admin') //is admin

 }

Constantele le punem intr-un enum

enum Role {ADMIN, READ\_ONLY, AUTHOR}

si folosim ca un obiect!!

const person2 = {

  name: 'Max',

  age: 30,

  hobbies: ['Sports', 'Cooking'],

  role: Role.ADMIN

};

if(person2.role === Role.ADMIN) {

  console.log('is admin') //is admin

 }

enum Role {ADMIN = 'ADMIN' , READ\_ONLY = 100, AUTHOR = '200'}

## 20. The "any" Type

let favoriteActivities2: any[];

favoriteActivities2 = ['sport', 5 , [2, 5]]

## 21. Union Types

function combine(n1:number, n2:number) {

  const result = n1 + n2

    return result;

}

const combinedAges =  combine(30,26);

console.log('combinedAges:', combinedAges);

Asa nu va functiona

const combinedNames =  combine('Max', 'Gigi');

refactorizam

function combine(n1:number | string, n2:number | string) {

  let result;

  if(typeof n1 === 'number' && typeof n2 === 'number') {

    result = n1 + n2

  } else {

    result = n1.toString() + n2.toString();

  }

  //!!

  return result;

}

const combinedAges =  combine(30,26);

console.log('combinedAges:', combinedAges);

const combinedNames =  combine('Max', 'Gigi');

console.log('combinedNames:', combinedNames);

## 22. Literal Types

function combine(n1:number | string, n2:number | string, resultType:'as-number' | 'as-text') {

  let result;

  if(typeof n1 === 'number' && typeof n2 === 'number' || resultType === 'as-number' ) {

    result = +n1 + +n2

  } else {

    result = n1.toString() + n2.toString();

  }

  //!!

  return result;

}

const combinedAges =  combine(30,26, 'as-number');

console.log('combinedAges:', combinedAges);

const combinedStringAges =  combine('30', '26', 'as-number');

console.log(combinedStringAges)

const combinedNames =  combine('Max', 'Gigi', 'as-text');

console.log('combinedNames:', combinedNames);

## 23. Type Aliases / Custom Types

type Combinable =  number | string ;

type ConversionDescription = 'as-number' | 'as-text'

function combine(n1: Combinable, n2:number | string, resultType: ConversionDescription ) {

  let result;

  if(typeof n1 === 'number' && typeof n2 === 'number' || resultType === 'as-number' ) {

    result = +n1 + +n2

  } else {

    result = n1.toString() + n2.toString();

  }

  //!!

  return result;

}

## 24. Type Aliases & Object Types

Type aliases can be used to "create" your own types. You're not limited to storing union types though - you can also provide an alias to a (possibly complex) object type.

For example:

type User = { name: string; age: number };

const u1: User = { name: 'Max', age: 30 }; // this works!

This allows you to avoid unnecessary repetition and manage types centrally.

For example, you can simplify this code:

//Alias

type User = { name: string; age: number };

const u1: User = { name: 'Max', age: 30 }; // this works!

//For example, you can simplify this code:

function greet(user: { name: string; age: number }) {

  console.log('Hi, I am ' + user.name);

}

function isOlder(user: { name: string; age: number }, checkAge: number) {

  return checkAge > user.age;

}

//Simplify

function greet(user: User) {

  console.log('Hi, I am ' + user.name);

}

function isOlder(user: User, checkAge: number) {

  return checkAge > user.age;

}

Quiz 2: Core Types & Concepts

**Which of the following snippets could be simplified by using an enum type?**

1. **const ROLE\_ADMIN = 0;**
2. **const ROLE\_AUTHOR = 1;**

**Will the following code throw a compilation error?**

1. **type User = {name: string; age: number};**
2. **const u1: User = ['Max', 29];**

Correct! **The "User" type clearly wants an object with a "name" and an "age" property**. NOT an array.

**Will this code make it through compilation?**

1. **type Product = {title: string; price: number;};**
2. **const p1: Product = { title: 'A Book', price: 12.99, isListed: true }**

**Will this code make it through compilation?**

1. **type User = { name: string } | string;**
2. **let u1: User = {name: 'Max'};**
3. **u1 = 'Michael';**

This code is fine. The union type allows either an object (with a "name" property) OR a string. You can switch values how often you want.

## 25. Function Return Types & "void"

function add(n1: number, n2:number) {

 return n1 + n2

}

function printResult(num:number) {

  console.log('Result ' + num)

}

printResult(add(5,12));.

## 26. Functions as Types

//26. Functions as Types

let combineValues;

combineValues = add;

console.log(combineValues(8,8))

let combineValues: Function;



## 27. Function Types & Callbacks

//27. Function Types & Callbacks

function addAndHandle(n1:number, n2: number, cb:(num:number) => void) {

  const result = n1 + n2

  cb(result)

}

addAndHandle(10, 20, (result) =>{

  console.log(result)

})

## Quiz 3: Functions & Types

**Will this code compile?**

1. **function sendRequest(data: string, cb: (response: any) => void) {**
2. **// ... sending a request with "data"**
3. **return cb({data: 'Hi there!'});**
4. **}**
6. **sendRequest('Send this!', (response) => {**
7. **console.log(response);**
8. **return true;**
9. **});**

That's correct. As you learned, callback functions can return something, even if the argument on which they're passed does NOT expect a returned value.

**What's the idea behind a "function type"?**



**Which code snippet is better (i.e. which code should you write)?**

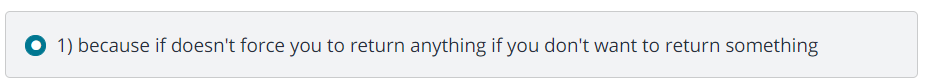
**1)**

1. **function sayHi(): void {**
2. **// ...**
3. **}**

**OR**

**2)**

1. **function sayHi(): undefined {**
2. **// ...**
3. **}**



## 28. The "unknown" Type

 let userInput: unknown;

 let userName: string

 userInput = 5;

 userInput = 'Max'

 //error

 userName = userInput;

error TS2322: Type 'unknown' is not assignable to type 'string'.

//error

   //userName = userInput;

   //corect

   if(typeof userInput === 'string') {

     userName = userInput;

   }

## 29. The "never" Type

function generateError(message: string, code: number):void {

     throw{message: message, errorCode: code}

  }

   const result = generateError('An error occured', 500);

   console.log(result)



  function generateError(message: string, code: number):never {

     throw{message: message, errorCode: code}

   }

   const result = generateError('An error occured', 500);

   console.log('R:', result)

## 30. Wrap Up

## 31. Useful Resources & Links

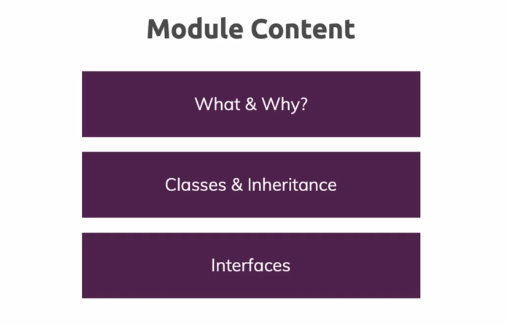
Attached you find all the code snapshots for this module - you also find them attached to individual lectures throughout this module.

These links might also be interesting:

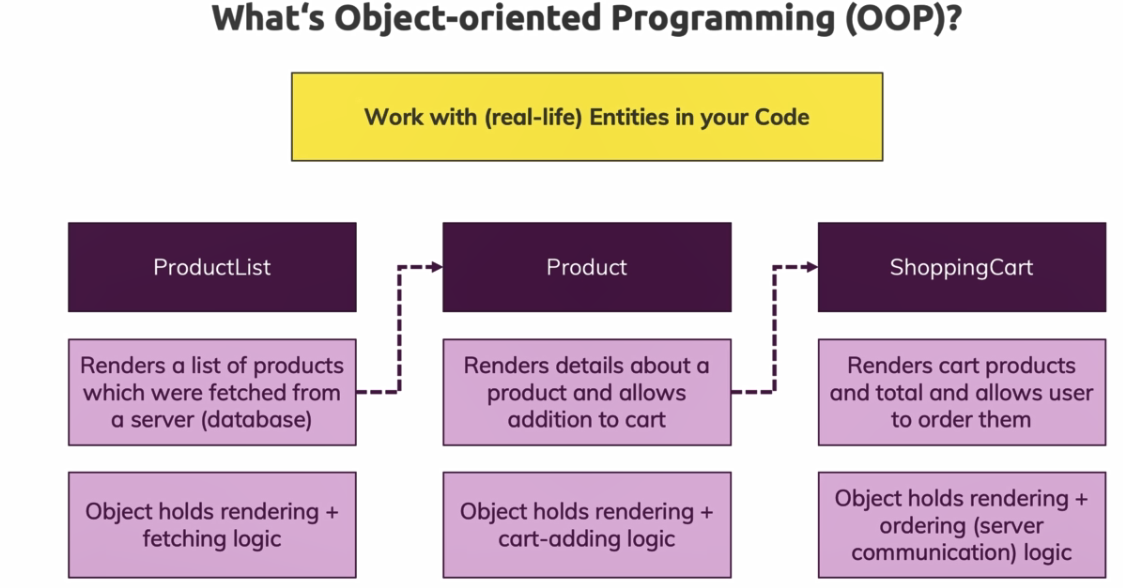
* Official TypeScript Docs:
* <https://www.typescriptlang.org/docs/handbook/basic-types.html>

# Section 5: Classes & Interfaces

## 56. Module Introduction



## 57. What are Classes?



## 58. Creating a First Class

// Code goes here!

class Department {

  name:string;

  constructor(n:string) {

    this.name = n;

  }

}

const accounting = new Department('Development')

console.log(accounting)

## 59. Compiling to JavaScript

## 60. Constructor Functions & The "this" Keyword

class Department {

  name:string;

  constructor(n:string) {

    this.name = n;

  }

  describe() {

    console.log('Department ' + this.name)

  }

}

const development = new Department('Development')

console.log(development)

development.describe();

//nu este executata in alt obiect

const devCopy = {describe: development.describe}

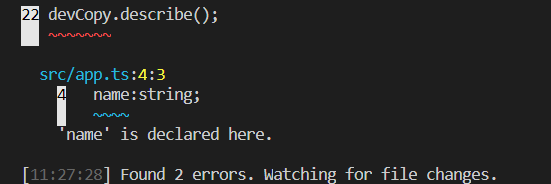
devCopy.describe();

  describe(this:Department) {

    console.log('Department ' + this.name)

  }

Error



**Asa functioneaza!!**

const devCopy = {name: 's', describe: development.describe}

devCopy.describe();

## 61. "private" and "public" Access Modifiers

Adaugam inca doua metode!

class Department {

  name:string;

  employees: string[] = []

  constructor(n:string) {

    this.name = n;

  }

  describe(this:Department) {

    console.log('Department ' + this.name)

  }

  addEmployee(employee:string) {

    this.employees.push(employee);

  }

  printEmployeeInformation() {

    console.log(this.employees.length);

    console.log(this.employees)

  }

//61. "private" and "public" Access Modifiers

development.addEmployee('Ghita');

development.addEmployee('Vasile');

development.printEmployeeInformation();

//se poate adauga si asa, ceea ce nu e bine

development.employees[2] = 'Lina'

development.printEmployeeInformation();

Trebuie sa facem o metoda private!

### Private!

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/Class_fields>

Adaugam private la array-ul employee

class Department {

  name:string;

  private employees: string[] = []

si vom avea erroarea

//se poate adauga si asa, ceea ce nu e bine

development.employees[2] = 'Lina'

development.printEmployeeInformation();

src/app.ts:43:13 - error TS2341: **Property 'employees' is private and only accessible within class 'Department'**.

43 development.employees[2] = 'Lina'

[11:43:22] Found 2 errors. Watching for file changes.

## 62. Shorthand Initialization

class Department {

  private employees: string[] = []

  constructor(private id: string, public name:string) {

  }

  describe(this:Department) {

    console.log('Department ' + this.id + ' ' + this.name)

  }

  //61. "private" and "public" Access Modifiers

  addEmployee(employee:string) {

    this.employees.push(employee);

  }

  printEmployeeInformation() {

    console.log(this.employees.length);

    console.log(this.employees)

  }

}

const development = new Department('d1', 'Development')

console.log(development)

development.describe();

//61. "private" and "public" Access Modifiers

development.addEmployee('Ghita');

development.addEmployee('Vasile');

development.printEmployeeInformation();

## 63. "readonly" Properties

constructor(private readonly id: string, public name:string) {

  }

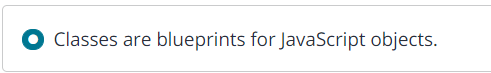
<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain>

## Quiz 4: Class Basics

Question 1:

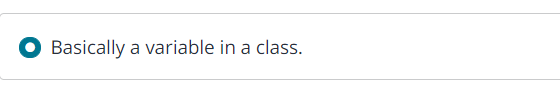
**What's the core idea behind classes?**

That's correct! You define how objects (which are instantiated based on classes) should look like



Question 2:

**What's a class property?**



Question 3:

**What's the idea of the private and public modifiers?**

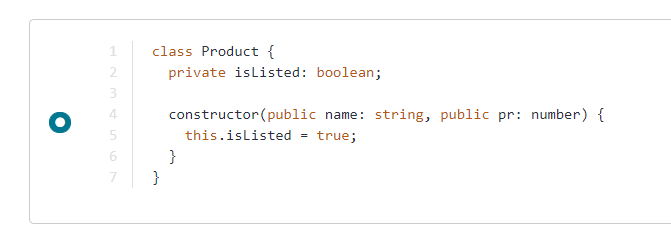


That's correct! You ensure that a "private" property is only accessible from inside a class (e.g. from inside a class method).

Question 4:

**How can you shorten the following code in the best possible way?**

1. **class Product {**
2. **title: string;**
3. **price: number;**
4. **private isListed: boolean;**
6. **constructor(name: string, pr: number) {**
7. **this.title = name;**
8. **this.price = pr;**
9. **this.isListed = true;**
10. **}**
11. **}**



## 64. Inheritance

//64 Inheritance

class ITDepartment extends Department {

  constructor(id:string, public admins: string[]) {

    super(id, 'IT')

    this.admins = admins

  }

}

const development = new ITDepartment('d2',['Ionel'])

console.log(development)

development.describe();

development.addEmployee('Ionela');

development.addEmployee('Marin');

development.printEmployeeInformation();

class AccountingDepartment extends Department {

  constructor(id:string, private reports: string[]) {

    super(id, 'Account')

  }

  addReport(text:string) {

    this.reports.push(text);

  }

  printReports() {

    console.log(this.reports)

  }

}

const accounting2 = new AccountingDepartment('d2', [])

accounting2.addReport('somefhing went wrong');

accounting2.printReports();

## 65. Overriding Properties & The "protected" Modifier

**Rewrite the method!**

### Protected

**Schimbam din private in protected**

class Department {

  protected employees: string[] = []

**este ok sa avem metoda**

addEmployee(name:string) {

    if(name === 'Max') {

      return

    }

    this.employees.push(name)

  }

## 66. Getters & Setters

class AccountingDepartment extends Department {  
 private lastReport: string;  
   
 //getter  
 get mostRecentReport() {  
 if(this.lastReport) {  
 return this.lastReport;  
 }  
 throw new ***Error***('No report found')  
 }  
 constructor(id:string, private reports: string[]) {  
 super(id, 'Account')  
 this.lastReport = reports[0]  
 }