https://www.udemy.com/understanding-typescript/

Typescript Udemy

Section_1_Getting_Started

Lesson 2 What is TypeScript?

Lesson 4 Installing TypeScript

Lesson 5 Using TypeScript

Lesson 6 Setting Up Workspace

Example code:

Section_2_Using_Types

2.1 String,Number,Boolean,Array,tuples,enum,any

2.2 Functions

2.3 Object, alias type

2.4 Union, typeof, never, nullable

2.5 Sample exercise (convert from js to ts)

Section_3_Understanding_typescript_compiler

Lesson 31 Compilation Behaviour

Lesson 32 Changing compilation behaviour

Lesson 36 Typescript2.0 compilation

Section_4_typescript_and_ES6(already covered)

Section_5_Classes_for_Objects

Lesson 57 Create a class

Lesson 57 Methods and Access modifiers

Lesson 58 Inheritance

Lesson 61 Getters & Setters

Lesson 62 Static properties and methods(helper class)

Lesson 63 Abstract Classes

Lesson 64 Private Constructors & Singletons

Lesson 65 ReadOnly properties

Section_6_NameSpaces_modules

Lesson 70 NameSpaces

Lesson 75 Modules

Section_7_ContractWork_with_Interfaces

Lesson 82 Interfaces Basics

Lesson 84 Interface Properties & Methods

Lesson 85 Interfaces and Classes

Lesson 86 Interfaces and function types

Lesson 87 Interfaces inheritance

Section_8_Generics

Lesson 91 Generic Basics

Lesson 92 Better Generic

Lesson 93 Built In Generics

Lesson 94 Array

Lesson 95 Using Generic types

Lesson 96 Using Generic Classes

Lesson 100 using generic types for miniproject

Section_9_Decorators

Lesson 101 Decorators Basics

Lesson 102 Class Decorators

Lesson 103 Decorator Factories

Lesson 104 Creating a useful decorator + Multiple decorators

Lesson 108 Method Decorator

Lesson 109 Property Decorators

Lesson 110 Parameter Decorator

Section_10_Third Party Libraries

Lesson 112 Installing JQuery

Lesson 116 Translating JavaScript to typescript

Section_11_Typescript_Workflows

Lesson 121 Using “tsc” and tscongfig file

Lesson 122 TypeScript resolves Files with tsconfig.json

Lesson 124 TypeScript with Gulp

Lesson 125 TypeScript with webpack

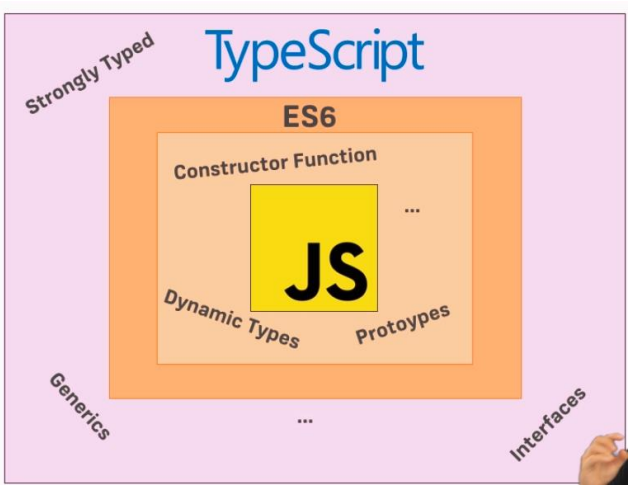
Section_12_Typescript+React

Lesson 128 Setting up Project Files

Lesson 132 Creating ReactJs code with Typescript

Section_1_Getting_Started

https://www.typescriptlang.org/



Lesson 2 What is TypeScript?

Typescript is a typed superset of Javascript which is compiled into plain javascript.

The reason why it is good to use it, that you can **specify exact types for the arguments, or end results**, so you can easily debug if there is a mismatch.

Lesson 4 Installing TypeScript

Install it globally

```
npm -g install typescript
```

Lesson 5 Using TypeScript

1.Create a typescript file with **.ts** extension

App.ts

2.Run the typescript to compile your code into plain javascript. (in Node)

Tsc app.ts →(it will generate a js file with the same name)

3.Include it in the html

```
<script src="app.js"></script>
```

Lesson 6 Setting Up Workspace

Setup a "**little-server**" to run a small server, to enable us to use modules as well, serves multiple files.

1. npm init (for creating a package.json)
2. npm install lite-server –save-dev
3. In package.json add a code to be executed in the "start" command

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "start": "lite-server"
},
```

4. Letting typescript knows that how and what should be converted to javascript

```
tsc --init
```

It will create a **tsconfig.json** file which will tell typescript what and how should it compile

Example code:

```
class Greeter {
// We define the greeting property to be a string
    greeting: string;
//Argument of the constructor function should be a string as well
    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}

let greeter = new Greeter("world");

let button = document.createElement('button');
button.textContent = "Say Hello";
button.onclick = function() {
    alert(greeter.greet());
}

document.body.appendChild(button);
```

Section_2_Using_Types

Types are only checked by typescript during compilation. In the compiled js file, there will be no types.

2.1 String,Number,Boolean,Array,tuples,enum,any

//Javascript has dynamic types, typescript has static types

// You define types once and use it.

string

```
let myName: string = 'Max';
// myName = 28;
```

number, no differentiantion between integer, float

```
let myAge: number = 27;
// myAge = 'Max';
```

boolean

```
let hasHobbies: boolean = false;
// hasHobbies = 1;  --> will fail!
```

assign types

```
let myRealAge: number;
myRealAge = 27;
// myRealAge = '27';
```

array, (array of any type of element)

```
let hobbies: any[] = ["Cooking", "Sports"];
hobbies = [100];
// hobbies = 100;
```

tuples //(mixed types)

```
let address: [string, number] = ["Superstreet", 99];
```

enum `//Make numbers, more expressive, an object with properties`

```
enum Color {
    Gray, // 0
    Green = 100, // 100
    Blue = 2// 2
}
let myColor: Color = Color.Blue;
console.log(myColor);
```

any `//You can assign Any kind of types, no compilation check`

```
let car: any = "BMW";
console.log(car);
car = { brand: "BMW", series: 3};
console.log(car);
```

2.2 Functions

Defining returned value, `//should return a string`

```
function returnMyName(): string {
    return myName;
}
console.log(returnMyName());
```

void `//Will not return anything`

```
function sayHello(): void {
    console.log("Hello!");
}
```

defining argument types

```
function multiply(value1: number, value2: number): number {
    return value1 * value2;
}
// console.log(multiply(2, 'Max'));
console.log(multiply(10, 2));
```

function as types `// We define a ES6 function to be an actual type as well`

```
let myMultiply: (a: number, b: number) => number;
// myMultiply = sayHello;
// myMultiply();
myMultiply = multiply;
console.log(myMultiply(5, 2));
```

2.3 Object, alias type

Define objects

```
//We can define, how an object should look like
let userData: { name: string, age: number } = {
    name: "Max",
    age: 27
};
// userData = {
//     a: "Hello",
//     b: 22
// };
```

complex object

```
// Should be an object, with a data & output property
// data should be an array of numbers
// output should be a function which return an array of numbers
// and it's argument should be a type of boolean
let complex: {data: number[], output: (all: boolean) => number[]} = {
    data: [100, 3.99, 10],

    output: function (all: boolean): number[] {
        return this.data;
    }
};
// complex = {};  //this would not satisfy our complex type
```

type alias

```
//adding a name to our type as an alias so we can reference it
type Complex = {data: number[], output: (all: boolean) => number[]};

let complex2: Complex = {
    data: [100, 3.99, 10],
    output: function (all: boolean): number[] {
        return this.data;
    }
};
```

2.4 Union, typeof, never, nullable

union types `//This can be number or string as well.`

```
let myRealRealAge: number | string = 27;
myRealRealAge = "27";
// myRealRealAge = true;
```

check types with **typeof**

```
let finalValue = 30;
if (typeof finalValue == "number") {
    console.log("Final value is a number");
}
```

```
/*Typescript 2.0*/
```

never, it never returning anything

```
function neverReturns():never{
    throw new Error('an error');
}
```

nullable

```
// in tsconfig.json, you can define not to be able to assign null to a value,
if you are not especially define it in the beginning
//     "strictNullChecks":true

// Nullable types, shouldn't be able to assign null to a type
let cantBeNull:number = 12;
cantBeNull = null;           // Will give an error
let canAlsoBeNull;          //by default it will be undefined
canAlsoBeNull = null;
//null can be assigned, to undefined, it is an exception


let canBeNull : number | null = 12;
canBeNull = null;
```

2.5 Sample exercise (convert from js to ts)

```
let bankAccount = {
    money: 2000,
    deposit(value) {
        this.money += value;
    }
};

let myself = {
    name: "Max",
    bankAccount: bankAccount,
    hobbies: ["Sports", "Cooking"]
};

myself.bankAccount.deposit(3000);

console.log(myself);
```

```
type BankAccount = { money: number, deposit: (val: number) => void };
// define our custom complex object

let bankAccount: BankAccount = {
    money: 2000,
    deposit(value: number): void {    //every variable should have a type
        this.money += value;
    }
};

//here we are using the object once
let myself: { name: string, bankAccount: BankAccount, hobbies: string[] } = {
    name: "Max",
    bankAccount: bankAccount,
    hobbies: ["Sports", "Cooking"]
};


myself.bankAccount.deposit(3000);


console.log(myself);
```

Section_3_Understanding_typescript_compiler

Official documentation for ts configuration

https://www.typescriptlang.org/docs/handbook/tsconfig-json.html

http://www.typescriptlang.org/docs/handbook/compiler-options.html

Lesson 31 Compilation Behaviour

Typescript compiler, will give you an error, but it will compile ts to js anyway

Lesson 32 Changing compilation behaviour

- **module**: in which js format should it compile,
- **target**: to which js standard,
- **noEmitOnError**: doesn't compile if there is an error, (default false)
- **Exclude**: what we do not want to compile,
- **sourceMap: true** -> the sourcemap file will be created as well (we will be able include ts files in chrome developer "**Sources**" tab)
- **noImplicitAny**: **true** -> Every variable should be defied! Any type won't be assigned as default.
- **strictNullChecks: true ->** you can't assign null to a variable, only if you especially declare to be it null as well.
- **noUnusedParameters true** → Will check if every parameter is used in a function

tsconfig.json (we can change it in this config file)

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "es5",
    "noImplicitAny": false,
    "sourceMap": false,
    "noEmitOnError":true
  },
  "exclude": [
    "node_modules"
  ]
}
```

Lesson 36 Typescript2.0 compilation

(Can check if the defined variable is used or not) (`strictNullChecks`)

```
// Check if the variable is being used or not
//here if strictNullChecks = true,
//we will get an error, since if isTrue=false;
// result variable of type number will not be used, prior returning it
function controlMe(istrue:boolean){
                let result:number;
                if(istrue){
                        result=12;
                }
                return result;
}
```

(Check if all of the parameters have been used or not) (`noUnusedParameters`)

Section_4_typescript_and_ES6

Everything has been covered in other courses.

Typescript and ES6 compatible stuff

http://kangax.github.io/compat-table/es6/

Section_5_Classes_for_Objects

Lesson 57 Create a class

```
// 0. define class
class Person {
    name: string;               //object property
    private type: string;       //only typescript, only accessible within this
object
    protected age: number = 27; //only TS, private + inherited classes can
access it
//will be used to create this object,
// public: can be access outside of the object, with defining this variable in
the constructor
    constructor(name: string, public username: string) {
        this.name = name;
    }
}
// 1. instantiate the class
const person = new Person("Max", "max");
```

Lesson 57 Methods and Access modifiers

```
//public method
    printAge() {
        console.log(this.age);
        this.setType("Old Guy");
    }
//private, only accessible inside
    private setType(type: string) {
        this.type = type;
        console.log(this.type);
    }
```

Lesson 58 Inheritance

```
// Inheritance
//Use all of the code specified in the person class
class Max extends Person {
    // name = "Max";
    constructor(username: string) {
        super("Max", username);  //will use all of the parent's
methods, properties
        this.age = 31;
    }
}
const max = new Max("max");
console.log(max);
console.log(max.type);  //will cause an error since type is a
private property of “Person” class, which can be only accessed in
the person object
```

Lesson 61 Getters & Setters

```
// Getters & Setters
// Control the _species property with getter + setter
class Plant {
    private _species: string = "Default";
    get species() {
        return this._species;
    }
    set species(value: string) {
        if (value.length > 3) {
            this._species = value;
        } else {
            this._species = "Default";
        }
    }
}


let plant = new Plant();
console.log(plant.species);
plant.species = "AB";
console.log(plant.species);
plant.species = "Green Plant";
console.log(plant.species);
```

Lesson 62 Static properties and methods(helper class)

You can access the object properties without creating an instance of the object/class

```
// Static Properties & Methods
class Helpers {
    static PI: number = 3.14;
    static calcCircumference(diameter: number): number {
        return this.PI * diameter;
    }
}
console.log(2 * Helpers.PI);
console.log(Helpers.calcCircumference(8));
```

Lesson 63 Abstract Classes

Abstract classes can't be instantiated, they need to be inherited and then they can be used. It is not needed for them to be instantiated.

Usage: They create a **basic layout, setup** for any other object, which should be instantiated.

Object's function can be abstract as well.

```
abstract class Project {
    projectName: string = "Default";
    budget: number = 1000;
//abstact method, as well
    abstract changeName(name: string): void;
    calcBudget() {
        return this.budget * 2;
    }
}
class ITProject extends Project {
//Which will be inherited and used
    changeName(name: string): void {
        this.projectName = name;
    }
}

let newProject = new ITProject();
console.log(newProject);
newProject.changeName("Super IT Project");
console.log(newProject);
```

Lesson 64 Private Constructors & Singletons

Singleton: can have only 1 instance all the time, no more!
Private Constructors: Can be executed only within the object's static methods.

```
class OnlyOne{
// SO the class's instance property is private, it can be accessible only within the class
// and it is static as well, so you do not need an instance to use it
    private static instance:OnlyOne;
//the constructor function is private as well, so it can be access only within the object
    private constructor(public name:string){}
//this is the only exposed function, which will create an instance of itself, if there is no instance already existing
    static getInstance(){
        if(!OnlyOne.instance){
            OnlyOne.instance = new OnlyOne('The Only One');
        }
    return OnlyOne.instance;
    }
}


let wrong = new OnlyOne('The Only One'); //Can't access it outside the object
let right = OnlyOne.getInstance;
//This can be accessed only, and it will create keep only 1 instance
```

Lesson 65 ReadOnly properties

```
    public readonly name:string;
//the constructor function is private as well, so it can be access only within the object
    private constructor(public name:string){
        this.name = name;
    }
```

Section_6_NameSpaces_modules

You can set object properties to read only, instead of using setters.

```
    public readonly name:string;
//the constructor function is private as well, so it can be access only within the object
    private constructor(public name:string){
        this.name = name;
    }
```

Lesson 70 NameSpaces

NameSpaces: is a set of symbols that are used to organize objects of various kinds, so that these objects may be referred to by name.

(grouping global variables in an object)

Lesson 75 Modules

Modules: separating the js files, and exporting+importing the desired objects, functions, variables.

We need a module loader, which loads all kind of module formats. "**SystemJs**"

0. install systemjs module

```
npm install systemjs –save
```

1. Add this systemjs file

```
 <script src="node_modules/systemjs/dist/system.js"></script>
```

2. Import our code

```
<script>
    // set our baseURL reference path
    SystemJS.config({
        baseURL: '/',
        defaultJSExtensions: true
    });
    SystemJS.import('app.js');
</script>
```

Module imports, relative/absolute paths

```
import * as Circle from "./math/circle"; //relative path
import {Component} from '@angular/core'; //absolute path
```

Section_7_ContractWork_with_Interfaces

Lesson 82 Interfaces Basics

Interface: It is a contract made by an object, that guarantees to have a certain functions, properties, methods.

So here, we are creating a `NamedPerson interface, a contract.`

`When we define person:NamedPerson, the person object accept the interface contract and it has to have a firstName object property!`

```
interface NamedPerson {
    firstName: string;
}

function greet(person: NamedPerson) {
    console.log("Hello, " + person.firstName);
}
function changeName(person: NamedPerson) {
    person.firstName = "Anna";
}

const person: NamedPerson = {
    firstName: "Max",
    age: 27
};

greet(person);
changeName(person);
greet(person);
```

Lesson 84 Interface Properties & Methods

1. Interfaces can have optional arguments, or dynamic properties with dynamic types (unknown ones)
2. Also it can have methods defined
3. Object literals are scanned much more than the const declared objects

```
// person argument has to have the firstName propery
1.
interface NamedPerson {
    firstName: string;
    age?: number;                   //optional argument
    [propName: string]: any;
//we will unknown named properties, with unknown types
2.
    greet(lastName: string): void;
//a method, with a string argument and a void return value
}

function greet(person: NamedPerson) {
    console.log("Hello, " + person.firstName);
}

function changeName(person: NamedPerson) {
    person.firstName = "Anna";
}

const person: NamedPerson = {
    firstName: "Max",
    hobbies: ["Cooking", "Sports"],
    greet(lastName: string) {
        console.log("Hi, I am " + this.firstName + " " + lastName);
    }
};
3
//When we pass object literals, it gets checked more than the created object
// So it will see that the age is not defined in the interface.
// greet({firstName: "Max", age: 27});
```

Lesson 85 Interfaces and Classes

```
interface NamedPerson {
    firstName: string;
    age?: number;                   //optional argument
    [propName: string]: any;
//we will unknown named properties, with unknown types
    greet(lastName: string): void; //
}


//We can create a class, which implements the NamedPerson interface,
// So we have to create all of the required properties, methods defined in the
interface
class Person implements NamedPerson {
    firstName: string;
    lastName: string;


    greet(lastName: string) {
        console.log("Hi, I am " + this.firstName + " " + lastName);
    };
}


const myPerson = new Person();
myPerson.firstName = "Maximilian";
myPerson.lastName = "Anything";
greet(myPerson);
myPerson.greet(myPerson.lastName);
```

Lesson 86 Interfaces and function types

```
// Function Types
//Whatever uses this interface, must be a function of this defined type
interface DoubleValueFunc {
    (number1: number, number2: number): number;
}

let myDoubleFunction: DoubleValueFunc;
myDoubleFunction = function (value1: number, value2: number) {
    return (value1 + value2) * 2;
};

console.log(myDoubleFunction(10, 20));
```

Lesson 87 Interfaces inheritance

```
//interface can inherit from other interfaces
interface AgedPerson extends NamedPerson {
    age: number;
}


const oldPerson: AgedPerson = {
    age: 27,
    firstName: "Max",
    greet(lastName: string) {
        console.log("Hello!");
    }
};


console.log(oldPerson);
```

Interfaces are not compiled at all! It is just there to check your code during compilation.

Section_8_Generics

Lesson 91 Generic Basics

Generic: is basically a way to write code which is really dynamic.

TypeScript supports parameterized types, also known as generics, which can be used in a variety of scenarios. For example, you can create a function that can take values of any type, but during its invocation, in a particular context, you can explicitly specify a concrete type.

```
/ Simple Generic
// the following funciton is a kind of generic function, because this function
// can get executed in any kind of data.
function echo(data: any) {
    return data;
}

console.log(echo("Max"));
console.log(echo(27));
console.log(echo({name: "Max", age: 27}));
```

Lesson 92 Better Generic

(Can define the type <>, or it will recognize by the type of the argument)

```
// Better Generic <> -> this will tell typescript that we are talking about a
generic
//So the function, will know which kind of data should be the end result
function betterEcho<T>(data: T) {
    return data;
}
//The types will be determined by the type of the argument.
// If the types support . notation + additional properties like .length like
this
// it will allow and handle it as well.
console.log(betterEcho("Max").length);
//it will recognize a number in the argument, so the type will be number
console.log(betterEcho(27));
//We can Explicitly     determine the type of the generic function
console.log(betterEcho<number>(27));
//error, typeMismatch, typescript will recognize that this is not the defined
type
console.log(betterEcho<number>("27"));
console.log(betterEcho({name: "Max", age: 27}));
```

Lesson 93 Built In Generics

Array is a built in generics

```
const testResults: Array<number> = [1.94, 2.33];
testResults.push(-2.99);
testResults.push('string'); //will give an error



console.log(testResults);
```

Lesson 94 Array

(can define dynamic type of array)

```
// here we dynamically determine the array with T, and we define that we
// are waiting for an array, of the later defined type
function printAll<T>(args: T[]) {
    args.forEach((element) => console.log(element));
}
printAll<string>(["Apple", "Banana"]);
printAll<number>([100, 200]);
```

Lesson 95 Using Generic types

```
// const typeAssignment (<T>(data: T) => T)
// echo2 is the name of the constant
// T is the dynamic type, which will be the type of the argument and the
result as well
// bettercho is the generic function, which will be assigned to the echo2
variable
//since, everything is dynamically typed, it is reusable
const echo2: <T>(data: T) => T = betterEcho;
console.log(echo2<string>("Something"));



function betterEcho<T>(data: T) {
    return data;
}
```

Lesson 96 Using Generic Classes

(We can dynamically use a class, constraining or defining types which can be used in the classes)

```
// Generic Class
// extends --> we can tell, which types can be used in this generic types
//So we can are constrained to use either number or string type.
class SimpleMath<T extends number | string, U extends number | string> {
    baseValue: T;
    multiplyValue: U;
    calculate(): number {
// + is used in order, to convert string numbers to number numbers
        return +this.baseValue * +this.multiplyValue;
    }
}


const simpleMath = new SimpleMath<string, number>();
simpleMath.baseValue = "10";
simpleMath.multiplyValue = 20;
console.log(simpleMath.calculate());
```

Lesson 100 using generic types for miniproject

Create a generic Map (an Object like an Array, but instead with Key-Value Pairs). The key will always be a string.

Let's keep it simple and only add the following methods to the Map:

```
setItem(key: string, item: T) // should create a new key-value pair

getItem(key: string) // should retrieve the value of the provided key
clear() // should remove all key-value pairs
printMap() // should output key-value pairs
```

The map should be usable like shown below:

```
const numberMap = new MyMap<number>();
numberMap.setItem('apples', 5);
numberMap.setItem('bananas', 10);
numberMap.printMap();

const stringMap = new MyMap<string>();
stringMap.setItem('name', "Max");
stringMap.setItem('age', "27");
stringMap.printMap();
```

```
class MyMap<T> {  //<T> is the type of the stored object elements
//this is the private variable, which stored unkown array of string keys, and
T type values, which is basically an empty object
    private map: {[key: string]: T} = {};
//Add a new key item to the object
    setItem(key: string, item: T) {
        this.map[key] = item;
    }
//retrive the item
    getItem(key: string) {
        return this.map[key];
    }
//make this an empty object
    clear() {
        this.map = {};
    }
//You can loop over the keys in the map object
    printMap() {
        for (let key in this.map) {
            console.log(key, this.map[key]);
        }
    }
}


const numberMap = new MyMap<number>();
numberMap.setItem("apples", 10);
numberMap.setItem("bananas", 2);
console.log(numberMap.getItem("apples"));
numberMap.printMap();
numberMap.clear();
numberMap.printMap();
```

Section_9_Decorators

https://www.typescriptlang.org/docs/handbook/decorators.html

Lesson 101 Decorators Basics

Decorators: are functions, that you can attach to methods, properties, classes and then work with them or transforming them.
Enables you to add extra functionality simply just adding a decorator to the class.

Lesson 102 Class Decorators

```
// This is the created decorator, which will be attached to the class
// By default, these function gets 1 argument, the constructor function of the attached class
// Typscript feature not ES6!!
// So this decorator, will simply console log the class's constructor function to which it has been added
function logged(constructorFn: Function) {
    console.log(constructorFn);
}


@logged
class Person {
    constructor() {
        console.log("Hi!");
    }
}
```

Tsconfig.json

"**experimentalDecorators**": true → means decorators are experimental, and you accept it, so no warning should be visible during compilation regarding this

Lesson 103 Decorator Factories

Factory

- You can overwrite the argument list according to your preference
- Based on the input, the decorator functions can call other functions to be attached to the current class

```
function logged(constructorFn: Function) {
    console.log(constructorFn);
}

// Factory
// it will determine based on the input, to return the previously written
// logged function, which will be added to the Car class, or return null
function logging(value: boolean) {
    return value ? logged : null;
}


@logging(true)
class Car {


}
```

Lesson 104 Creating a useful decorator + Multiple decorators

- You can add new functions to the prototype of the decorated class
- This is the way to call it (<any>**className**).**functionName**();
- You can add multiple decorators

```
function logged(constructorFn: Function) {
    console.log(constructorFn);
}
// Factory
function logging(value: boolean) {
    return value ? logged : null;
}
// Advanced
// You can attach new functions to the prototpye of the classes, which are
decorated
function printable(constructorFn: Function) {
    constructorFn.prototype.print = function () {
        console.log(this);
    }
}
@logging(false)
@printable
class Plant {
    name = "Green Plant";
}
const plant = new Plant();
// This is the proper way to call those functions which added to the prototype
// with the decorator
(<any>plant).print();
```

Lesson 108 Method Decorator

- **Target**, **propName**, **descriptor** are the main variable provided by typescript
- You can access the methods metadata and modify it
- Make it editable or not

```
// will help us modify methods
// target: targeted method
// propName: the property name
// descriptor:   https://developer.mozilla.org/en-
US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty
// it is basically a description of the property if it should be configurable,
enumerable etc.
function editable(value: boolean) {
    return function (target: any, propName: string, descriptor:
PropertyDescriptor) {
        descriptor.writable = value;
    }
}


class Project {
    @editable(false)
    calcBudget() {
        console.log(1000);
    }
}
const project = new Project("Super Project");
project.calcBudget();
project.calcBudget = function () {
    console.log(2000);
};
project.calcBudget();
console.log(project);
```

Lesson 109 Property Decorators

- You can change the descriptor property of every Object property using decorators
- However, if you are changing the descriptor.writable property to false, it will be pretty much invisible, not accessible

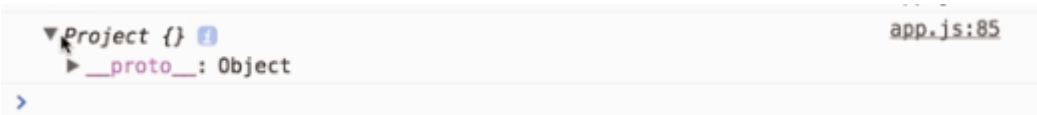
```
// We are returning the new descriptor of the property
// https://developer.mozilla.org/en-
US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty
function overwritable(value: boolean) {
    return function (target: any, propName: string): any {
        const newDescriptor: PropertyDescriptor = {
            writable: value
        };
        return newDescriptor;
    }
}

class Project {
    @overwritable(false)
    projectName: string;


    constructor(name: string) {
        this.projectName = name;
    }
}

console.log(project);
```

The property can’t be seen



Lesson 110 Parameter Decorator

- It is possible to access and execute some code on the parameters where this kind of decorator is inserted

```
// we can access the paremeters and execute some code on it.
// target: the paremeters
// methodName: the name of the method, in which we are added the decorator
// paramIndex: parameter index
function printInfo(target: any, methodName: string, paramIndex: number) {
    console.log("Target: ", target);
    console.log("methodName: ", methodName);
    console.log("paramIndex: ", paramIndex);
}

class Course {
    name: string;

    constructor(name: string) {
        this.name = name;
    }
    printStudentNumbers(mode: string, @printInfo printAll: boolean) {
        if (printAll) {
            console.log(10000);
        } else {
            console.log(2000);
        }
    }
}
const course = new Course("Super Course");
course.printStudentNumbers("anything", true);
course.printStudentNumbers("anything", false);
```

This is what will be logged out.

Section_10_Third Party Libraries

Lesson 112 Installing Jquery

0. Install jquery

```
npm install –save jquery
```

1. Import the library with systemJs

Index.html

```
<script>
    // set our baseURL reference path
    SystemJS.config({
        map: {
            "jQuery": "node_modules/jquery/dist/jquery.min.js"
        },
        baseURL: '/',
        defaultJSExtensions: true
    });
    SystemJS.import('app.js');
</script>
```

2. You can include it in you ts files (name if the import as in the map should be the same

App.ts

```
import "jQuery";
$("#app").css({"background-color": "green"});
```

Lesson 116 Translating javascript to typescript

You can write it on your own
https://www.typescriptlang.org/docs/handbook/declaration-files/introduction.html

Or you can download the available ones
http://definitelytyped.org/directory/projects.html

0. Install jquery types (Typescript 2.0)

```
npm install --save @types/jquery
```

And it is ready, you can use jquery freely in your code!

Section_11_Typescript_Workflows

Lesson 121 Using “tsc” and tscongfig file

When we run **tsc** command in node, we are creating the js files from ts.

```
tsc –w
```

It is in watch mode so it will autocompile if there is any changes.

Lesson 122 TypeScript resolves Files with tsconfig.json

https://www.typescriptlang.org/docs/handbook/tsconfig-json.html

Tscongif.json

- **Exclude:** -> won’t be compiled
- **Module:** -> in which format should the compilation be done
- **Target:** -> to which version of ecmascript
- **noImplicitAny:** -> null can be assigned to every variable

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "es5",
    "noImplicitAny": false,
    "sourceMap": false
  },
  "exclude": [
    "node_modules"
  ]
}
```

Lesson 124 TypeScript with Gulp

https://www.npmjs.com/package/gulp-typescript

0. Installing Gulp

```
npm install –save-dev gulp gulp-typescript
```

gulp-typscript

It is normal typescript just wrapped in gulp workflow.

1. Create a gulpfile.js

```
// Importing the dependecies

var ts = require("gulp-typescript");

var gulp = require("gulp");



//Will tell gulp, where is the tsconfig file is located, based on which
project can be created

var tsProject = ts.createProject("tsconfig.json");



gulp.task("typescript", function() {

  return tsProject.src()

    .pipe(ts(tsProject)) // compile all fo the ts file, taking into consideration
the tsconfig file

    .pipe(gulp.dest("")); // outputPatk

});

// Will watch any typescript files for changes

gulp.task("watch", function() {

  gulp.watch("*.ts", ["typescript"]);

});

//default task, when running gulp, run watch mode!

gulp.task("default", ["watch"]);
```

2. Add a build script to **package.json**

```
"scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "lite-server",
    "build": "gulp"
  },
```

```
npm run build
```

it will start gulp in watch mode, so if you change sth, it will be recompiled

```
npm start
```

If you restart your server, you will see the changes.

Lesson 125 TypeScript with webpack

0. Install dependencies

```
npm install –save–dev webpack ts-loader
```

ts-loader: typescript file compiler to use in wepback

1. Remove the systemJs scripts from **index.html** include bundle.js

```
<script src="bundle.js"></script>
```

This will be the file, where all of the js file will be compiled.

2. Remove unnecessary lines from **tsconfig.json**

```
{
  "compilerOptions": {
    "target": "es5",
    "noImplicitAny": false
  }
}
```

3. Creating **webpack.congif.json**

Entry: where all of the app starts
Output: where all of the js files should be compiled
Loaders: all of the file containing ts should be handled by ts-loader

```
module.exports = {
    entry: "./app.ts",
    output: {
        filename: "bundle.js"
    },
    module: {
        loaders: [
            { test: /\.ts$/, loader: "ts-loader" }
        ]
    }
};
```

4. Add new scripts to **package.json**
Build: will run webpack for **d**evelopment, in **w**atchmode
Build:prod: will create production files

```
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "lite-server",
    "build": "webpack -d --watch",
    "build:prod": "webpack -p"
  },
```

4.1 npm install –save–dev typescript (typescript has to be installed so ts-loader could use it)

5. Update the importing in **app.ts**

```
// importing js file, as $ sign
//this is the syntax, what webpack knows
import $ = require("jquery");
```

In order to run the webpack ensure, you made the following commands

Run "**npm** install" to install the required dependencies

Run "**typings**" to install the required typings (needs the "typings" package to be installed on your machine => "**npm install typings -g**")

Run "**npm run build**" to compile the TypeScript code and start Webpack Watcher

Run "**npm start**" to run the development server (lite-server)

Section_12_Typescript+React

Lesson 128 Setting up Project Files

0. Npm should handle all of the dependencies, create **package.json**

```
npm init
```

1. Install dependencies

```
npm install –save react react-dom
```

2. Adding the typescript definition types
   (**npm install typings -g**) if you don't have it installed gobally

```
typings install dt~react dt~react-dom –global –save
```

They will be references in the **typings.json file**

```
{
  "globalDependencies": {
    "react": "registry:dt/react#0.14.0+20160829191040",
    "react-dom": "registry:dt/react-dom#0.14.0+20160412154040"
  }
}
```

3. Installing webpack

```
npm install webpack ts-loader –save-dev
```

4. Having a server to serve our files

```
npm install -save little-server
```

4.5 Storing every react code in the **src** folder

5. Add a **webpack.config.json** file

- **Entry:** where the whole app starts
- **Output:** it is where the bundle should be created
- **Devtool:** source-map , to enable source map for debugging
- **Resolve:** extensions[], includes all of the extensions which webpack should know
- **Loaders:** determine which kind if file, with which kind of module will be compiled

```
module.exports = {
    entry: "./src/index.tsx",
    output: {
        filename: "./dist/bundle.js"
    },
    // Enable sourcemaps for debugging webpack's output.
    devtool: "source-map",

    resolve: {
        // Add '.ts' and '.tsx' as resolvable extensions.
        extensions: ["", ".ts", ".tsx", ".js"]
    },
    module: {
        loaders: [
 // All files with a '.ts' or '.tsx' extension will be handled by 'ts-loader'.
            { test: /\.tsx?$/, loader: "ts-loader" }
        ]
    }
};
```

Lesson 132 Creating ReactJs code with Typescript

Index.tsx

```
import * as React from "react";
import * as ReactDOM from "react-dom";

import { Home } from "./components/Home";

ReactDOM.render(<Home name="Max" age={27}/>, document.getElementById("app"));
```

Home.tsx

```
import * as React from "react";

// Being specification, which properties should have the create object
interface HomeProps {
    name: string;
    age: number;
}
//interface, state
//<HomeProps, {}>
export class Home extends React.Component<HomeProps, {}> {
    render() {
        return (
            <div>
             Hello there, {this.props.name}, you are {this.props.age}, right?
            </div>
        );
    }
}
```

Add start, build script to package.json

```
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "lite-server",
    "build": "webpack"
  },
```

npm install typescript –save–dev

Create a tsconfig file

```
Tsc init
```

```
{
    "compilerOptions": {
        "module": "commonjs",
        "target": "es5",
        "noImplicitAny": false,
        "sourceMap": false,
        "jsx": "react"
    },
    "exclude": [
        "node_modules"
    ]
}
```

So it will know how to handle jsx files.