<https://indepth.dev/>

Angular JS vs Angular > 2 - the first version of Angular without TypeScript vs Angular

**Typescript**

Typescript is a superset of JS, uses the .ts file extension:

| let isDone: boolean = false; let hex: number = 0xf00d; let binary: number = 0b1010; let octal: number = 0o744;  let message: string = '' let messages: string[] = ['Test 1','Test 2'] let messages: ['Test 1', number] = [Test 1', 2.3] |
| --- |

<https://www.typescriptlang.org/docs/handbook/intro.html>

tsc index.ts –target es6 - convert Typescript to JS based on a ECMA standard, транспилация

!!! you can dynamically change the type of a variable, but you can assign the value of a variable from different type to the variable, for which you want its type to be changed !!!

**Namespace in Angular:**

| namespace Vehicle {  export enum Car {  Peugeout = 'Message 1',  Citroen = 1  }   export enum Truck {  Ford = 'Ford',  RangeRover = 2  }  }  let carMessages: [number, Vehicle.Car] = [1, Vehicle.Car.Citroen] |
| --- |

**constructor property promotion is available in Typescript as well:**

| class Suv {   public model: string;   constructor(model: string) {  this.model = model;  }  } |
| --- |

= >

| class Suv {   constructor(public model: string) {   }  } |
| --- |

**method overriding**

| class Suv {   constructor(protected model: string) {   }   drive():string {  return `I am driving this ${this.model}`;  }  }  export class SmallerSuv extends Suv {   constructor(model: string, private size: number) {  super(model);  }   override drive(): string {  return super.drive() + ' called from child!'  }   park(): void {  console.log(`I am parking this ${this.model} in size ${this.size}`)  }  } |
| --- |

**we can set a custom type**

| // creating a custom type function printLabel(labelledObj: {label: string}){  console.log(labelledObj.label); } |
| --- |

**Generics** - primary used for data structures

function identity<T>(arg: T): T {

return arg;

}

let output = identity<string>('myString');

**!isNan(+arg)** - check if arg is number

!!! interface are possible in Typescript with the interface keyword and we can create objects of set interface as well as classes, however, we can not check if an object is instance of interface and interfaces are converted to basic JavaScript classes !!!

!!! native Javascript feature is # to define private or static properties and it is different from the Typescript:   
  
<https://stackoverflow.com/questions/59641564/what-are-the-differences-between-the-private-keyword-and-private-fields-in-types>

The private keyword in TypeScript is a compile time annotation. It tells the compiler that a property should only be accessible inside that class  
  
vs  
  
Private fields ensure that properties are kept private at runtime: !!!

**Install Angular:**

| npm install -g @angular/cli ng new some-app cd some-app //starting the server: ng serve  ng build |
| --- |

**Start Angular project:**

| [15:18:32] donetianpetkov@Donetian-Petkov-NEWs-MacBook-Pro [ ~/WebstormProjects/angular\_first/soft-uni-day-one ] $ ng serve  ✔ Browser application bundle generation complete.  Initial Chunk Files | Names | Raw Size vendor.js | vendor | 1.70 MB | polyfills.js | polyfills | 294.81 kB | styles.css, styles.js | styles | 173.24 kB | main.js | main | 47.69 kB | runtime.js | runtime | 6.53 kB |   | Initial Total | 2.21 MB  Build at: 2022-04-03T12:18:54.282Z - Hash: 19f21a7f6345c537 - Time: 11344ms  \*\* Angular Live Development Server is listening on localhost:4200, open your browser on http://localhost:4200/ \*\*   ✔ Compiled successfully. |
| --- |

**Build:**

| [15:01:27] donetianpetkov@Donetian-Petkov-NEWs-MacBook-Pro [ ~/WebstormProjects/angular\_first/soft-uni-day-one ] $ ng build  ✔ Browser application bundle generation complete. ✔ Copying assets complete. ✔ Index html generation complete.  Initial Chunk Files | Names | Raw Size | Estimated Transfer Size main.20a8ef08287cd70f.js | main | 116.45 kB | 34.88 kB polyfills.bd73d25360343036.js | polyfills | 33.03 kB | 10.64 kB runtime.20c563868d71cc52.js | runtime | 1.06 kB | 606 bytes styles.ef46db3751d8e999.css | styles | 0 bytes | -   | Initial Total | 150.54 kB | 46.12 kB  Build at: 2022-04-03T12:24:06.435Z - Hash: 06132fb6ef214ebb - Time: 17633ms  [15:25:47] donetianpetkov@Donetian-Petkov-NEWs-MacBook-Pro [ ~/WebstormProjects/angular\_first/soft-uni-day-one ] $ cd dist/ [15:25:55] donetianpetkov@Donetian-Petkov-NEWs-MacBook-Pro [ ~/WebstormProjects/angular\_first/soft-uni-day-one/dist ] $ ls soft-uni-day-one [15:25:55] donetianpetkov@Donetian-Petkov-NEWs-MacBook-Pro [ ~/WebstormProjects/angular\_first/soft-uni-day-one/dist ] $ cd soft-uni-day-one/ [15:25:58] donetianpetkov@Donetian-Petkov-NEWs-MacBook-Pro [ ~/WebstormProjects/angular\_first/soft-uni-day-one/dist/soft-uni-day-one ] $ ls 3rdpartylicenses.txt index.html polyfills.bd73d25360343036.js styles.ef46db3751d8e999.css favicon.ico main.20a8ef08287cd70f.js runtime.20c563868d71cc52.js |
| --- |

**config file: angular.json** - how the project will be built

<https://stackblitz.com/> - online IDE for JS, Angular, React and etc  
  
<https://plnkr.co/> - online IDE for JS, Angular, React and etc

**tsconfig.json** - how the Typescript and the project will be compiled

**Component in Angular:**

| import { Component } from '@angular/core';  @Component({  selector: 'app-root',  templateUrl: './app.component.html',  styleUrls: ['./app.component.css'] }) export class AppComponent {  title = 'soft-uni-day-one'; } |
| --- |

**Manually Creating Components in Angular:**  
  
1. Create component ts in the src/app folder -> define component inside it:  
  
​​

| import { Component } from '@angular/core';  @Component({  selector: 'app-test',  template: `<div>Test</div>`,  styles: [`div {color:blue}`] }) export class TestComponent {  title = 'testing'; } |
| --- |

2. then include the component in the app.module.ts:

| import {TestComponent} from "./test.component";  @NgModule({  declarations: [  AppComponent,  TestComponent |
| --- |

3. and finally set the selector in the app.component.html or the template file where you want to use the component:

| <app-test></app-test> |
| --- |

**Automatically Creating Components:**

| [19:21:59] donetianpetkov@Donetian-Petkov-NEWs-MacBook-Pro [ ~/WebstormProjects/angular\_first/soft-uni-day-one ] $ ng generate component example  CREATE src/app/example/example.component.css (0 bytes) CREATE src/app/example/example.component.html (22 bytes) CREATE src/app/example/example.component.spec.ts (633 bytes) CREATE src/app/example/example.component.ts (279 bytes) UPDATE src/app/app.module.ts (467 bytes) |
| --- |

**ng g c example** - shorter syntax

!!! the ng g c finds the closest module to the newly created component and adds the component in the Declaration !!!

!!! Every Angular project has a module file (App.Module) where the Angular app is launched:

| import { NgModule } from '@angular/core'; import { BrowserModule } from '@angular/platform-browser';  import { AppComponent } from './app.component'; import {TestComponent} from "./test.component"; import { ExampleComponent } from './example/example.component';  @NgModule({  declarations: [  AppComponent,  TestComponent,  ExampleComponent  ],  imports: [  BrowserModule  ],  providers: [],  bootstrap: [AppComponent] }) export class AppModule { } |
| --- |

in the bootstrap: we set which components will be launched when the app is executed. We need to also specify in the index.html the selector for these components:  
  
**app.component.ts:**

| import { Component } from '@angular/core';  @Component({  selector: 'app-root',  templateUrl: './app.component.html',  styleUrls: ['./app.component.css'] }) export class AppComponent {  title = 'soft-uni-day-one'; } |
| --- |

->   
  
**index.html:**

| <body>  <app-root></app-root> </body> |
| --- |

!!!!

**interpolation is only for showing text in HTML - {{}}**

**if we need to use a value from the class we use square brackets - in other words we bind:**   
  
in the template / html file for the component:

<input [value]="title"> - title will be the title property in the class  
  
**events in Angular:**  
  
in the template / html file for the component:  
  
<button (click)="nameOfFunction()">Click Me</button> - when we click the button the nameOfFunction will be executed  
  
in the TS file for the component:

| **export class AppComponent {  nameOfFunction(): returnType {  // some functionality  } }** |
| --- |

**Directives in Angular**special commands for working within the HTML template - \*ngFor, \*ngIf

in the component ts file:

| users = [  {  name: 'Ivan',  age: 20  },  {  name: 'Naiden',  age: 30  },  {  name: 'Ivana',  age: 50  }  ] |
| --- |

in the template / html file:

| <ul>  <li \*ngFor="let user of users">  {{user.name}}  </li> </ul> |
| --- |

in the template / html file:

| <div \*ngIf="showText">VISIBLE</div>  <button (click)="toggleText()">{{showText ? 'HIDE TEXT' : 'SHOW TEXT'}}</button> |
| --- |

in the component ts file:

| showText = true;  toggleText(): void {  this.showText = !this.showText;  } |
| --- |

**the event in the callback functions is $event**

| <button (click)="toggleText($event)">{{showText ? 'HIDE TEXT' : 'SHOW TEXT'}}</button> |
| --- |

->

| toggleText(event: MouseEvent): void {  event.preventDefault();  this.showText = !this.showText;  } |
| --- |

**binding / adding CSS classes to elements:**in html:

| <div \*ngIf="showText" [class]="classes">VISIBLE</div> |
| --- |

in component:

| export class ExampleComponent implements OnInit {   classes=['test', 'test-1'];  in component css:  .test {  color:red; } |
| --- |

**conditional class - >** [class.special]="isSpecial"

if isSpecial in the class is true, the element will have CSS class special

**adding styles inline:**

| [style.color]="isSpecial ? 'red' : 'blue'" |
| --- |

**Templates variables**calling HTML elements in other elements - this is how we can take the values of input fields by clicking a button without a form:

| <input #inputElement type="text">  <button (click)="changeTitleHandler(inputElement.value)">Change Title</button>  <p>{{title}}</p> |
| --- |

**nullsafe operators**

| {{game?.title}} |
| --- |

!!!! not recommended to use two-way data binding - [(ngModel)] !!!  
  
**Lifecycle Hooks**

ngOnInit - after constructor before content  
ngAfterViewInIt  
ngOnDestroy   
ngOnChanges - when Input changes, we can call ngOnChanges(simpleChanges: SimpleChanges) { console.log(simpleChanges} to see the change, need for the reference to be changed  
ngDoCheck

Презентационни компоненти vs Container компоненти - display data vs process data

!!! we can set a property to a class without initializing it with the ! after the prop - user!: IUser   
  
Another alternative: user: IUser | undefined; !!!

**Passing input to child components:**  
  
**<app-user-list-item>:**ts:

| @Input() user!: IUser; |
| --- |

html:

| <span>{{user?.name}}</span> <span>{{user?.age}}</span> |
| --- |

->  
  
**<app-user-list>:**

ts:

| @Input() userArray: {name: string, age: number}[] = []; |
| --- |

html:

| <div id="container">   <app-user-list-item \*ngFor="let users of userArray" [user]="users"></app-user-list-item> </div> |
| --- |

->

**<app-root>:**

html:

<app-user-list [userArray]="users"></app-user-list>  
  
ts:

| users = [  {  name: 'Ivan',  age: 20  },  {  name: 'Naiden',  age: 30  },  {  name: 'Ivana',  age: 50  }  ] |
| --- |

**Passing Output to Parent Components:**  
  
same example:   
  
**<app-user-list>:**  
  
ts:

| @Output() addUser = new EventEmitter<IUser>();  addNewUser(userNameInput: HTMLInputElement, userAgeInput: HTMLInputElement): void {   const {value: name} = userNameInput;  const {valueAsNumber: age} = userAgeInput;   this.addUser.emit({name, age}); // triggering the addUser and emitting the name and age to the parent component - addUser is an event and in the parent component we have addNewUserHandler, which handles this event   userAgeInput.value = '';  userNameInput.value = '';   } |
| --- |

html:

| <div>  <input #userNameInput type="text">  <input #userAgeInput type="number">  <button (click)="addNewUser(userNameInput, userAgeInput)">Add New User</button>  </div> |
| --- |

**<app-root>:**

ts:

| addNewUserHandler(newUser: IUser){  this.users.push(newUser);  } |
| --- |

html:

<app-user-list [userArray]="users" (addUser)="addNewUserHandler($event)"></app-user-list>

**Debugging in Angular** - setting a debugger after the line we want to be executed:

| constructor() {  console.log("I am in the constructor")  debugger;   } |
| --- |

Reloading the page it will spawn the debugger on the browser  
  
**zone.js** - gathers all of the async functions from the browser and monitors them - provides a mechanism, called zones, for encapsulating and intercepting asynchronous activities in the browser (e.g. setTimeout, , promises); set in the polyfills.tsl ; uses monkey patching

**changeDetection** - set in the @Component - changes between Default rendering and OnPush rendering (disables rendering on reference changes) ->   
  
 changeDetection: ChangeDetectionStrategy.OnPush  
  
**SOLID principles** -   
  
S - Single-responsiblity Principle: A class should have one and only one reason to change, meaning that a class should have only one job.

O - Open-closed Principle: Objects or entities should be open for extension but closed for modification.

L - Liskov Substitution Principle: every subclass or derived class should be substitutable for their base or parent class.

I - Interface Segregation Principle: A client should never be forced to implement an interface that it doesn’t use, or clients shouldn’t be forced to depend on methods they do not use.

D - Dependency Inversion Principle: Entities must depend on abstractions, not on concretions. It states that the high-level module must not depend on the low-level module, but they should depend on abstractions.

<https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design>

**Dependency Injection** - implementation of the Dependency Inversion; ensure that instead of initializing manually dependencies for a class, instead we tell the class which are its dependencies and Angular initiates them in the constructor.   
  
**Instead of:**

| public class Laptop {   public battery: Battery;   public videoCard: VideoCard;   constructor() {   this.battery = new Battery('Acer battery');   this.videoCard = new VideoCard('Nvidia 960 GTX');   }  } |
| --- |

**we can use**

| public class Laptop {  constructor(public battery: Battery, public videoCard: VideoCard) {}  } |
| --- |

!!! We need to include the Battery and the VideoCard in the providers array in the module TS file like:

providers: [{  
 provide: Battery,  
 useClass: Battery  
 }]!!!

**useValue** - we can set a custom string, object, function and etc

!!! **Services:** Components should not fetch or save data directly - this is why we use services. Components should focus on presenting data and delegate data access to a service. Services are a great way to share information among classes that don't know

about each other and avoid code duplication !!!

!!! we can also include providers in the Component’s declaration and in the main.ts !!!

**@Inject** - we can inject primitive types and etc in the dependency injector through an InjectionToken:   
  
**in the module**:

| export const myStringInjectionToken = new InjectionToken('myString');  {  provide: myStringInjectionToken,  useValue: 'this is the value'  } |
| --- |

**in the class where we want to use the injectiontoken:**

| constructor(@Inject(myStringInjectionToken) myString: string) {   console.log(myString);   } |
| --- |

**@Injectable** - we can use this declaration in the class, which we want to provide in the app, without including it in the providers array in the module:

| @Injectable({  providedIn: 'root' }) export class UserService |
| --- |

!!! the recommended way to declare dependencies it is to add them to the providers array - we may use the following shorter syntax:  
  
**instead of**

| providers: [{  provide: Battery,  useClass: Battery  }] |
| --- |

**we can use:**   
  
providers: [Battery] !!!

**Observables** - similar to promises, which can be sync or async, an observable represents a stream, or source of data that can arrive over time. You can create an observable from nearly anything, but the most common use case in RxJS is from events. This can be anything from mouse moves, button clicks, input into a text field, or even route changes; can be defined and used with a dollar sign at the end of a stream variable - streamOne$

| import { of } from 'rxjs'; import { map } from 'rxjs/operators'  of(1000).pipe(map(x => x + 100)).subscribe((x) => {  console.log(x); }); // the same thing as:   //Promise.resolve(1000).then(x => x+100).then(console.log); |
| --- |

**Cold vs Hot Observables** - independent subscriptions vs shared subscription

tap vs map - map returns a value to the stream, tap does not and thus, it can be used for a side effect - such as void functions  
  
!!! fake REST api for generating API streams - <https://jsonplaceholder.typicode.com/> !!!

!!! imdb API - <https://www.omdbapi.com/?i=tt3896198&apikey=3dbf0674> !!!

!!! turn JSON output to object manually - search-replace of the double quotes (ctrl + R):  
  
**"(.\*)":**to  **$1:**   
  
!!!

**make a HTTP request** - >   
  
**app.module.ts:**

| import { HttpClientModule} from "@angular/common/http";  imports: [  HttpClientModule  ], |
| --- |

**the component where the HTTP request is made:**

| import {HttpClient, HttpClientModule} from "@angular/common/http";   loadUsers(search: string = '') {   const query = search ? `?email\_like=${search}` : '';   return this.http.get<IUser[]>(`https://jsonplaceholder.typicode.com/users${query}`)   } |
| --- |

**in the component where the data from the HTTP request is processed:**   
  
 this.userService.loadUsers().subscribe(users => this.users = users);

**Modules** - NgModules are containers for a cohesive block of code dedicated to an application domain, a workflow, or a closely related set of capabilities They can contain components, service providers, and other code files whose scope is defined by the containing NgModule. They can import functionality that is exported from other NgModules, and export selected functionality for use by other NgModules.  
  
**ng g m core** - generating a module  
  
we must include the module in the root / app module:

| imports: [  BrowserModule,  HttpClientModule,  CoreModule,  SharedModule,  UserModule  ] |
| --- |

!!! If we want to use components, services, etc from one module in another module we must include the components in the Exports array in the first module and include only the module in the Imports array in the second module !!!

**Routing**:

**Example:**  
**1) app-routing.module.ts:**

| import { Routes, RouterModule } from '@angular/router'; import { AboutComponent } from './about/about.component'; import { UserListComponent } from './user/user-list/user-list.component';  const routes: Routes = [  {  path: '',  pathMatch: 'full',  redirectTo: '/user-list'  },  {  path: 'user-list',  component: UserListComponent  },  {  path: 'about',  component: AboutComponent  } ]  export const AppRoutingModule = RouterModule.forRoot(routes); |
| --- |

**2) app.module.ts:**

| **imports: [ AppRoutingModule** |
| --- |

**3) app.component.html:**

| <div id="content">   <router-outlet></router-outlet>  </div> |
| --- |

**4) header.component.html:**

| <nav>  <a routerLink="/user-list">User List</a>  <a routerLink="/about">About Us</a>  </nav> |
| --- |

**5) module, where the header component has been set:**

| imports: [ RouterModule |
| --- |

1) We first define the routes in a custom Routing module  
2) We include the Routing module in the root / app module  
3) We set router-outlet in the component where the content of the routes will be displayed / rendered  
4) In the header component we set a tags with routerLink to build navigation buttons to the routes

5) we import the default RouterModule in the module where the header component has been set.

**Not Found Page:**

1) ng g c not-found

2) app-routing.module.ts:

| {  path: '\*\*',  component: NotFoundComponent  } |
| --- |

**making a tag active:**

html:

| <a routerLink="/user-list" routerLinkActive="active">User List</a>  <a routerLink="/about" routerLinkActive="active">About Us</a> |
| --- |

css file:

| a.active {  color:red; } |
| --- |

**SwitchMap** - The main difference between switchMap and other flattening operators is the cancelling effect. On each emission the previous inner observable (the result of the function you supplied) is cancelled and the new observable is subscribed. You can remember this by the phrase switch to a new observable.   
  
switchMap is one of the most useful RxJS operators because it can compose Observables from an initial value that is unknown or that change. Conceptually, it is similar to chaining then functions with Promises, but operates on streams (Promises resolve once). Example: You have an Observable of a userID, then use it to \*switch\* to an HTTP request of all posts owned by that user.

The switchMap operator returns an observable that emits items based on applying a function that you supply to each item emitted by the source observable where that function returns an inner observable. Each time it observes one of these inner observables, the output observable begins emitting the items emitted by that inner observable.

**!!!! Does not return an observable, but subscribes to itself and returns the items;**

**thus, can be used to combine two HTTP requests !!!!**

| this.activatedRoute.params.pipe(  switchMap(({id}) => this.userService.loadUser(id))  ).subscribe(user => {  this.user = user;  }) |
| --- |

Another variant with additional action in pipe:

| this.activatedRoute.params.pipe(  tap(() => this.user = undefined),  switchMap(({id}) => this.userService.loadUser(id))  ).subscribe(user => {  this.user = user;  }) |
| --- |

!!! switchMap will cancel any previous requests upon new request !!!

**subRouting** -

/test  
/test/one

/test/two

->

| const routes: Routes = [  {  path: 'test',  component: MainComponent,  children: [  {  path: '',  pathMatch: 'full',  redirectTo: '/test/one'  },  {  path: 'one',  component: OneComponent  },  {  path: 'two',  component: TwoComponent  }  ]  } ] |
| --- |

!!! Ensure that a module is only initialized once (through Singleton):

| constructor(@Optional() @SkipSelf() coreModule: CoreModule) {   if (coreModule) {  throw new Error('Core module should be imported once!');  }   } |
| --- |

!!!!

**Guards -** Angular route guards are interfaces provided by angular which when implemented allow us to control the accessibility of a route based on condition provided in class implementation of that interface.

**canActivate** - Interface that a class can implement to be a guard deciding if a route can be activated. If all guards return true, navigation continues. If any guard returns false, navigation is cancelled. If any guard returns a UrlTree, the current navigation is cancelled and a new navigation begins to the UrlTree returned from the guard.

Example: checking if id has been provided to /user-detail/ URI

1) Routing Module:

| {  path: 'user-detail/:id',  component: UserDetailComponent,  canActivate: [ParamsActivate],  data: {  paramsActivate: ['id'],  paramsActivateRedirectURL: '/user-list'  }  } |
| --- |

2) Component Module:

| providers: [  ParamsActivate  ] |
| --- |

3) params.activate.ts

| import { Injectable } from "@angular/core"; import { ActivatedRouteSnapshot, CanActivate, Router, RouterStateSnapshot, UrlTree} from "@angular/router"; import {Observable} from "rxjs";  @Injectable()  export class ParamsActivate implements CanActivate {   constructor(private router: Router) {   }   canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean | UrlTree | Observable<boolean | UrlTree> | Promise<boolean | UrlTree> {   const {data: {paramsActivate, paramsActivateRedirectUrl }} = route;   if (!paramsActivate || !Array.isArray(paramsActivate) || paramsActivate.length === 0) {  return true;  }   const hasAllRouteParams = paramsActivate.reduce((acc, curr) => {  return !!route.params[curr] && acc  }, true);   if (hasAllRouteParams) {  return true;  }   return this.router.parseUrl(paramsActivateRedirectUrl || '/');   } } |
| --- |

<https://codeburst.io/understanding-angular-guards-347b452e1892>

!!! **FontAwesome for Angular:**   
  
<https://www.npmjs.com/package/@fortawesome/angular-fontawesome>

!!!

**turn element to a boolean** - !!this.user, !!number, etc

**Check what platform you are on (browser, etc)** - > PLATFORM\_ID:

<https://angular.io/api/core/PLATFORM_ID>  
  
<https://github.com/angular/angular/blob/main/packages/common/src/platform_id.ts>

at module.ts - >

| export class CoreModule {   constructor(@Inject(PLATFORM\_ID) platformId: any) {  console.log(isPlatformBrowser(platformId));  }  } |
| --- |