Not defining type of variable causes it to have type any

**let** a;

a = 1;

a = true;

a = 'a'

Define enum in typescript :

**enum** Color {

    Red,

    Green,

    Blue

}

**let** backColor = Color.Blue;

tsc main.ts | node main.js

Intellisense doesn’t work in this situation :

we should use type assertion to have intellisense :

(note the type is not changed after assertion in runtime and still it is any type)

**let** message;

message = 'abc';

**let** endsWithC = (<string>message).endsWith('c')

**let** endsWithCAlt = (message as string).endsWith('c')

a way of defining function in typescript :

**let** drawpoint = (x,y) **=>** {

}

drawpoint2({

    x:'1', y:'2'

})

It’s how to define custom type in typrscript:

**interface** Point {

    x: number,

    y: number

}

**let** drawPoint3 = (point: Point) **=>** {

}

In interface we can have functions and because they can access the properties we should not specify the paramaters

**interface** Point2 {

    x: number,

    y: number,

    draw : () **=>** void

}

The cohesion concept in OOP tells us that we should have Point type and function in the same place call Class: (without this they are hanging in the air somehow)- Class groups properties and functions that are highly related

**class** Point {

    x: number;

    y: number;

    draw() {

*// ...*

    }

    getDistance(another: Point) {

*// ...*

    }

}

Object creation in typescript :

**let** pointObj: Point = new Point();

**let** pointObj2 = new Point();   *// looks better*

pointObj.x = 2;

pointObj.y = 3;

pointObj.draw();

in typescript we can not have multiple constructors so here is the solution for creating object without default constructor :

**class** Point {

    x: number;

    y: number;

**constructor**(x?: number, y?: number) {

        this.x = x || 0;

        this.y = y || 0;

    }

    draw() {

        console.log('X: ' + this.x + ', Y:' + this.y)

    }

    getDistance(another: Point) {

*// ...*

    }

}

In typescript we have private – public and protected access modifiers :

(by default properties are public)

**class** Point3 {

**private** x: number;

**private** y: number;

**constructor**(x?: number, y?: number) {

        this.x = x || 0;

        this.y = y || 0;

    }

    draw() {

        console.log('X: ' + this.x + ', Y:' + this.y)

    }

    getDistance(another: Point) {

*// ...*

    }

}

*// in js*

**let** log = **function**(message) {

    console.log(message)

}

*// in ts*

**let** doLog = (message) **=>** {

    console.log(message)

}

**let** doLog2 = (message) **=>** console.log(message)

**let** doLog3 = () **=>** console.log()

**class** Point4 {

**constructor**(**private** x: number, **private** y: number) {

    }

    draw() {

        console.log('X: ' + this.x + ', Y:' + this.y)

    }

    getDistance(another: Point4) {

*// ...*

    }

}

Properties in typeScript :

**class** Point4 {

**constructor**(**private** \_x: number, **private** \_y: number) {

    }

    draw() {

        console.log('X: ' + this.x + ', Y:' + this.\_y)

    }

    getDistance(another: Point4) {

*// ...*

    }

**get** x() {

        return this.\_x;

    }

**set** x(value) {

        if (value < 0) {

            throw new Error('value cannot be less than 0.')

        }

        this.\_x = value;

    }

}

new Point4(2,4).x = 4;

when we use export in class definition we can have a module :

export **class** Point5 {

**constructor**(**private** \_x: number, **private** \_y: number) {

    }

    draw() {

        console.log('X: ' + this.x + ', Y:' + this.\_y)

    }

    getDistance(another: Point4) {

*// ...*

    }

**get** x() {

        return this.\_x;

    }

**set** x(value) {

        if (value < 0) {

            throw new Error('value cannot be less than 0.')

        }

        this.\_x = value;

    }

}

import {Point5} from './point'

export **class** LikeComponent {

**constructor**(**private** \_likesCount: number, **private** \_isSelected: boolean) {

   }

   onClick() {

    this.\_likesCount += (this.\_isSelected) ? -1 : 1;

    this.\_isSelected = !this.\_isSelected

   }

**get** likesCount() {

      return this.\_likesCount;

   }

**get** isSelected(){

      return this.\_isSelected;

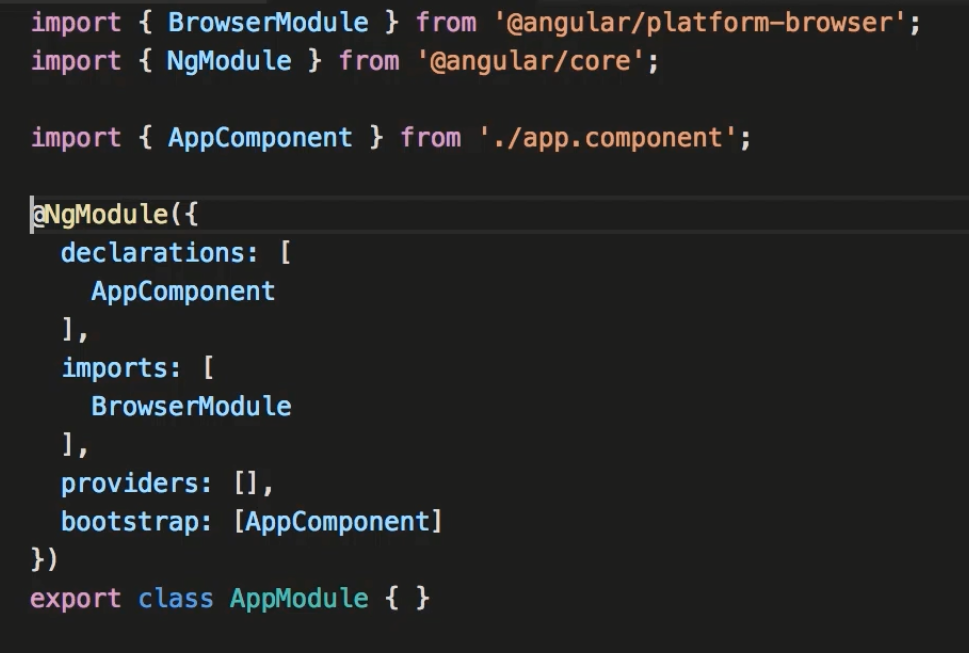
   }

}

We must register a component in a module to use it

Components are decorated with decorator functions @Component({}) and the templates and selector are assigned. In this way we can convert a plain typescript class to component

With decorator function @NgModule({}) we convert a plain typescript class to a angular module. In declaration property w specify components that are part of this module



Install auto import (steoates)

By using angular cli a component is created and added to module

ng g c course

this expression will be evaluated at runtime and the value of title field wii be inserted in the DOM

if the value of title field changes in the future Angular will automatically update the DOM

bind the view to a field in a component – this is called **string Interpolation**

<h2>{{ title }}</h2>

<h2>{{ "Title : " + title }}</h2> // simple javascript expression

<h2>{{ "Title : " + getTitle() }}</h2>

import { Component } from '@angular/core';

@Component({

  selector: 'app-course',

  templateUrl: './course.component.html',

  styleUrls: ['./course.component.css']

})

export **class** CourseComponent {

    title = 'List of courses'

    getTitle() {

      return this.title;

    }

  }

We use directives to manipulate a DOM element like removing or inserting or change a class of DOM element or it’s style (if we use directive that modifies DOM we should prefix it with asterisk)

<ul>

    <li \*ngFor="" ></li>

</ul>

<ul>

    <li \*ngFor="**let** course of courses" >

        {{course}}

    </li>

</ul>

We use a specific class called Service for implementing services in angular

In app module we should register any dependency that components in module are dependent upon

Angular will create a single object for entire module and will be injected (Singleton pattern)

ng g s courses

@NgModule({

  declarations: [

    AppComponent,

    CourseComponent

  ],

  imports: [

    BrowserModule

  ],

  providers: [

    CoursesService

  ],

  bootstrap: [AppComponent]

})

export **class** AppModule { }

here is how to consume the service via injection :

@Component({

  selector: 'app-course',

  templateUrl: './course.component.html',

  styleUrls: ['./course.component.css']

})

export **class** CourseComponent {

    title = 'List of courses'

    courses;

**constructor**(**private** service: CoursesService) {

      this.courses = service.getCourses();

    }

    getTitle() {

      return this.title;

    }

  }

This class in decorated with @Injectable function when it has dependency inside it’s constructors

@Injectable({

  providedIn: 'root'

})

export **class** MailService {

**constructor**(**private** service: LogService) {

    }

In case of component class it has injection inside component decorator function

ul>li will do the job

with Property binding we bind a property or attribute with a property or field in out component

 <img [src]="imageUrl" >

export **class** AuthorsComponent {

  authors;

  imageUrl : string = '/assets/angular.png'

**constructor**(**private** service: AuthorsService) {

    this.authors = service.getAuthors();

  }

**Property binding** is on way binding meaning if user changes DOM the underline fields or properties inside ts file wont get updated. But string interpolation is two way binding

<img  src= {{imageUrl}}  />

**Attribute binding**

There are some properties that html elements have but they don’t exist in DOM element inside memory and vice versa for example colSpan of td element is not in the DOM but in the html . Property binding in angular works for the properties that are in the DOM. We should prefix the propert with attr. In order it to work and not get error. This is called attribute binding

<table>

    <tr>

        <td colspan="attr.colSpan"></td>

    </tr>

</table>

export **class** CourseComponent {

    title = 'List of courses'

    courses;

    colSpan = 2;

in order to install bootstrap we use

npm install bootstrap –save

to install it in node\_modules folder and save it as dependency in package.json file

by listing the dependency in package.json when someone check out the project the dependency can be installed

in styles.css we import the reference of bootstrap folder in node\_module

@import "~bootstrap/dist/css/bootstrap.css";

body {

  padding:20px;

}

<button class="btn btn-primary">Save</button>

**Class binding** in angular : if value is true the class will be added if not the class will be removed

For this purpose we use a variation of property binding

<button class="btn btn-primary" [class.active]="isActive">Save</button>

export **class** CourseComponent {

    title = 'List of courses'

    courses;

    colSpan = 2;

    isActive = true;

**Style binding**

<button [style.backgroundColor]="isActive ? 'blue' : 'white'" >Save</button>

List of style binding can be found on w3school

Variation of property binding : class , attribute, style binding

**Event binding**

For handling the events raised by the DOM and instead of brackets we use parathesis

<button (click)="onSave()">Save</button>

Sometimes we want to get access to the event object that was raised in the event handler for example in the event object of a mouse movement we can access the x and y coordinates

We do it by sending $event parameter

<button (click)="onSave($event)">Save</button>

export **class** CoursesComponent {

  onSave($event) {

    console.log("Button was clicked", $event)

  }

}

**Event bubbling**

If a DOM event is fired is will bubble up all the way up and passess all the parrent elements

In this case after clicking the button the vent of the parent div will b fired.

<div (click)="onDivClicked()">

    <button (click)="onSave($event)">Save</button>

</div>

export **class** CoursesComponent {

  onDivClicked() {

    console.log("div clicked");

  }

  onSave($event) {

    console.log("Button was clicked", $event)

  }

}

We can stop propogation and bubbling up :

onSave($event) {

    $event.stopPropogation();

    console.log("Button was clicked", $event)

  }

Event filter

<input (keyup)="onKeyUp($event)" />

 onKeyUp($event) {

    if($event.keyCode === 13) console.log("Enter was pressed")

  }

There is a better way for doing this : (by eventfilter) :

<input (keyup.enter)="onKeyUp($event)" />

 onKeyUp($event) {

     console.log("Enter was pressed")

  }

**Template variable**

We can refere to the input field by target property of $event

console.log($event.target.value)

declare a variable in the template that refer to the inputfield

<input #email (keyup.enter)="onKeyUpNew(email.value)" />

  onKeyUpNew(email) {

    console.log(email)

 }

In property binding the direction of binding is from component in the view hence it is one way binding and changing it from the view doesn’t change the value in the log

<input [value]="email" (keyup.enter)="onKeyVer2()" />

export **class** CoursesComponent {

  email = "me@email.com";

 onkeyUpVer2() {

    console.log(this.email)

  }

}

**Two way binding with banana in the box**

<input [value]="email" (keyup.enter)="email = $event.target.value; onkeyUpVer2()" />

**Instead of above method :**

We can use a built-in angular directive called ngModule inside [()] for implementing two way binding

We should add FormsModule :

@NgModule({

  declarations: [

    AppComponent,

    CourseComponent,

    AuthorsComponent,

    CoursesComponent

  ],

  imports: [

    BrowserModule,

    FormsModule

  ],

<input [(ngModel)]="email" (keyup.enter)="onkeyUpVer2()"  />

Actually we bind to ngModule

**Pipes**

We use pipes to format data

Angular has some built-in types like uppercase and number(decimal)

{{course.title | uppercase}}

{{course.students | number }} <br/>

Pipes can be changed

{{course.title | uppercase  | lowercase }}

Number pipe can have parameter for number of integer and min and max number of precision

{{course.rating | number: '1.2-2'}}

{{course.rating | number: '2.2-3'}}

{{course.price | currency : 'AUD'}} <br/>

{{course.price | currency }} <br/>

{{course.price | currency : 'AUD': true:'3.2-2'}}

{{course.releaseDate | date:'shortDate'}}

When we Import the browser module it also import commonmodule

**Custom pipes**

Just like components we should register pipes in module

{{ text | summary }}

import {Pipe, PipeTransform} from '@angular/core'

@Pipe({

    name: 'summary'

})

export **class** SummaryPipe **implements** PipeTransform {

    transform(value: string, args?: any) {

        if(!value) {

            return null;

        }

        return value.substring(0, 50) + '...';

    }

}

@NgModule({

  declarations: [

    AppComponent,

    CourseComponent,

    AuthorsComponent,

    CoursesComponent,

    SummaryPipe

  ],

We can supply parameter in custom pipe :

{{ text | summary : 10}}

In angular we can change the parameter in method signature of an implemented interface

    transform(value: string, args?: any) {

transform(value: string, limit?: number) {

even we can have another parameters

transform(value: string, limit?: number, another?: number) {

Added to Github