

# Web services

## Angular



Samir Bellahsene  
NFC-eSIM Technology Manager at Orange/TGI/CEP  
[samir.bellahsene@orange.com](mailto:samir.bellahsene@orange.com)

January 2022

# Contents

## 1. Introduction

## 2. Environment and development tools

## 3. Main Concepts

### 3.1 Angular application Architecture

## 4. TypeScript

### 4.1 Variable declarations

### 4.2. Different types in TypeScript

### 4.3. Type Assertion

### 4.4. Arrow functions

### 4.5. Interfaces

### 4.6. Classes

### 4.7. Objects

### 4.8. Constructors

### 4.9. Access modifiers

### 4.10. Access modifiers in constructor parameters

### 4.11. Properties

### 4.12. Modules

## 5. Angular's building blocks

### 5.1. Modules

### 5.2. Components

### 5.3. Templates

### 5.4. Data binding

### 5.5. Pipes

### 5.6. Directives

### 5.7. Services and Dependency Injection (DI)

## 7. Exercise: make your Angular app

# #1 Introduction

# 1. Introduction

- ❑ Angular is an application design framework and development platform for creating single-page client applications.
- ❑ Developed by Google. The first versions called AngularJS (2009). Angular 2 (2016) or simply Angular refers to all versions from  $v \geq 2$ .
- ❑ It uses HTML and Typescript.
- ❑ Prerequisites : HTML, CSS and JavaScript with some of its latest standard such as classes and modules.
- ❑ Angular code samples provided in this course are written using TypeScript.

# #2 Environment and development tools

## 2. Environment development and tools (1/2)

- ❑ Install Node.js (<https://nodejs.org/en/download/>) → runtime environment for executing Javascript code outside the browser. (**node –version**)
- ❑ Node Package Manager (npm) will be installed with Node.js
- ❑ **npm install -g @angular/cli** → will install the Angular Command Line Interface (CLI) . (**npm –version**)
- ❑ **ng new my-app** → will create a workspace and the initial application →. The **ng new** command will prompt you for information about features to include in the initial app. Accept the defaults by pressing the Enter or Return key
- ❑ The Angular CLI installs the necessary Angular npm packages and other dependencies. The CLI creates a new workspace and a simple Welcome app, ready to run.
- ❑ Run the application: **cd my-app**. Then do **ng serve –open**  
The ng serve command launches the server and load your app on it, watches your files, and rebuilds the app as you make changes to those files. Use the option **--open (or just -o)** to automatically open your browser to **<http://localhost:4200/>**.
- ❑ CLI commands reference : <https://angular.io/cli>
- ❑ Install Visual Studio Code editor : <https://code.visualstudio.com/>
- ❑ Add VS code to the path to ease open it from the terminal (CLT + press Shift command +P for Mac users or Shift + CTRL + P) → **install code command in path**.
- ❑ From the directory of your Angular project write in Treminal “**code.**” → this will open VS code pointing to your project directory.

```
c:\>ng new my-app
? Do you want to enforce stricter type checking and stricter bundle budgets in the workspace?
  This setting helps improve maintainability and catch bugs ahead of time.
  For more information, see https://angular.io/strict Yes
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? CSS
CREATE my-app/angular.json (3623 bytes)
CREATE my-app/package.json (1196 bytes)
```

## 2. Environment development and tools (2/2)

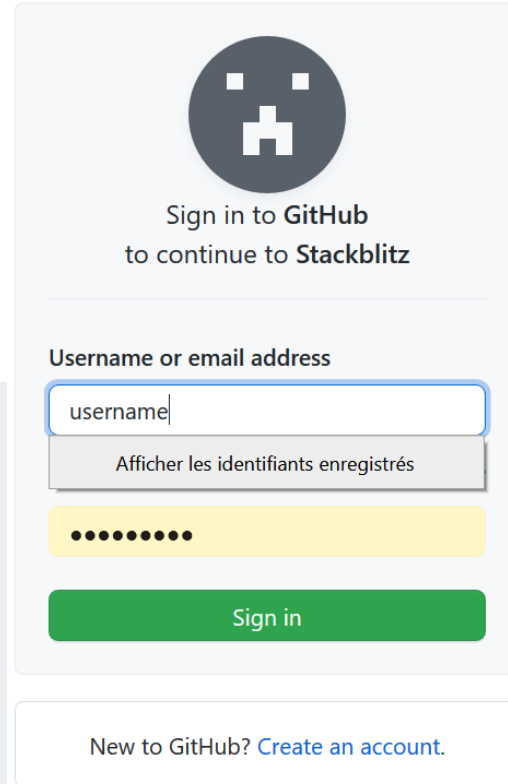
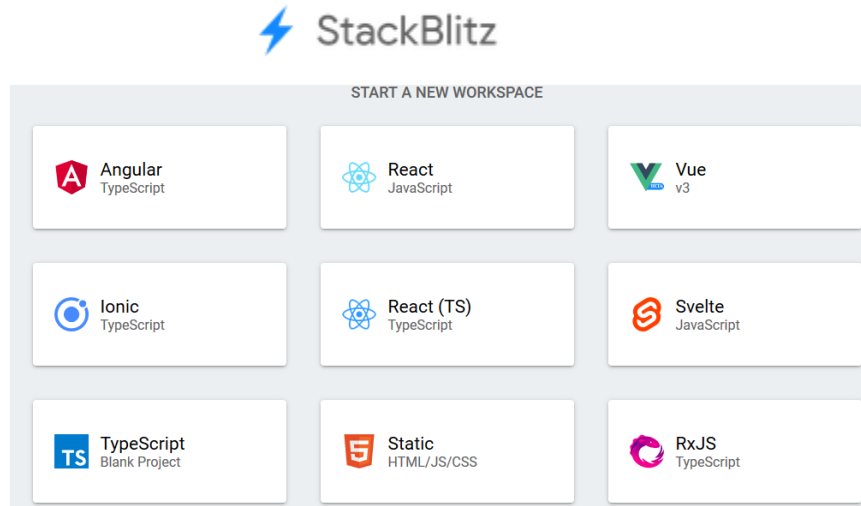
❑ **Online code editing : StackBlitz** (<https://stackblitz.com/>)

❑ **StackBlitz** provides another way to code your application online and from your browser. It provides the ability to create, edit and deploy fullstack applications.

❑ Need to sign in to GitHub to use StackBlitz

❑ Can be used to code with a large panel of web development frameworks (Angular, React, Vue.js, etc).

❑ Sponsored by the world known organizations & individuals (Google, Firebase, Ignite UI, etc).

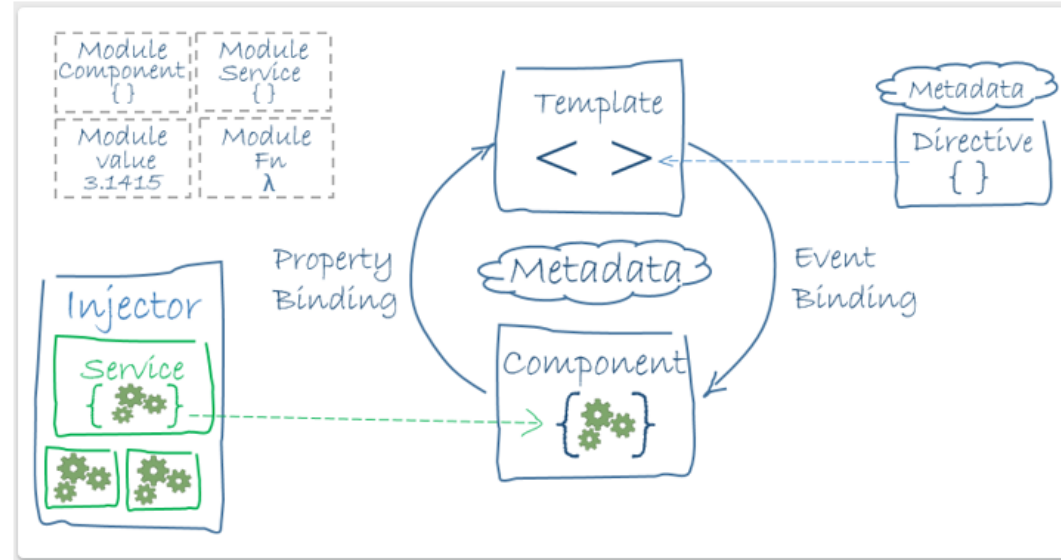


# #3 Main concepts



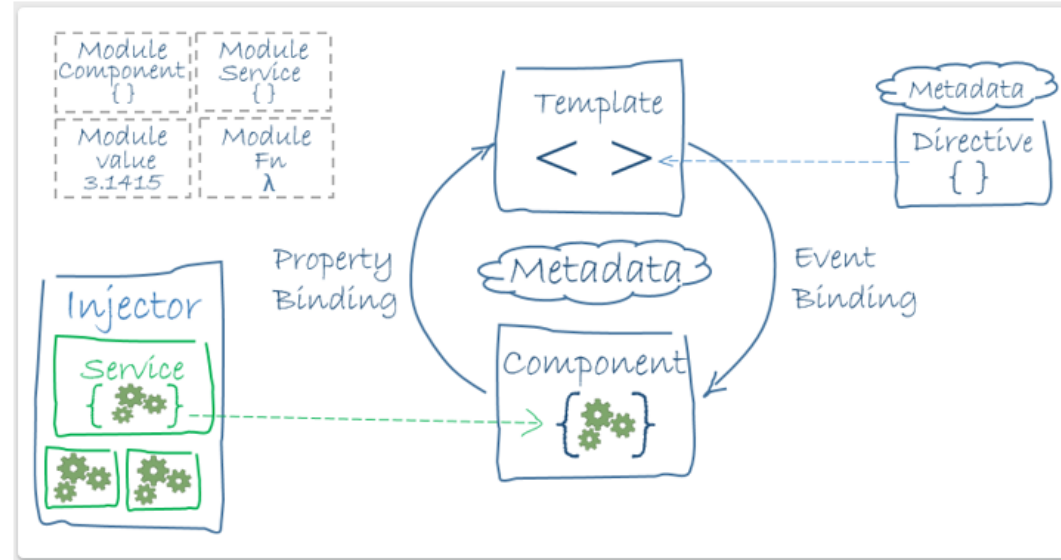
### 3.1. Angular application Architecture (1/3)

- ❑ The architecture of an Angular application relies on certain fundamental concepts.
- ❑ The basic building blocks of Angular app are **Angular components**.
- ❑ **Component** is a class that contains application data and programing logic, and is associated with an **HTML template** that defines a **view** to be displayed in a target environment.
- ❑ Components use **services**, which provide specific functionality not directly related to views. Service providers are **injected** into components as **dependencies** to make Angular code modular, reusable, and efficient.
- ❑ Angular components are organized into **NgModules**.
- ❑ **NgModule** is a block of code where is collected related codes into functional sets.

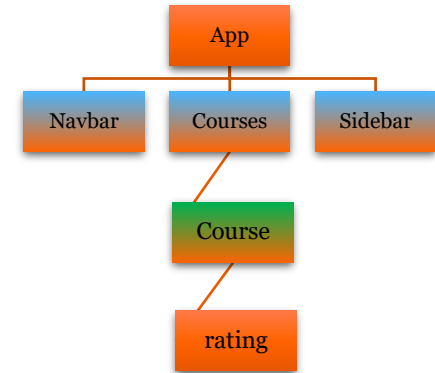
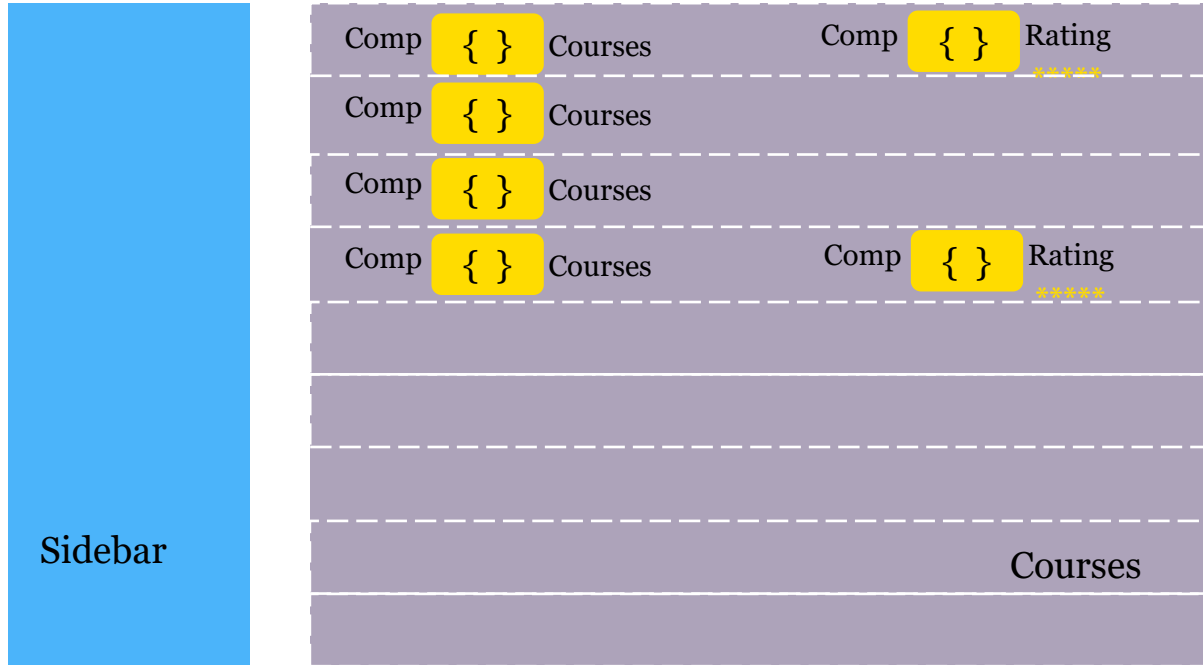


## 3.1. Angular application Architecture (2/3)

- Modules, components and services are classes that use **decorators** that mark their type and provide **metadata** that tells Angular how to use them.
- The **metadata** for a component class associates it with a template that defines a view.
- A template combines ordinary HTML with Angular **directives** and **binding markup (Property Binding & Event Binding)** that allow Angular to modify the HTML before rendering it for display.
- The **metadata** for a service class provides the information Angular needs to make it available to components through **dependency injection (DI)**.
- Components define many views, arranged hierarchically. Angular provides the **Router service** to define the navigation paths among these views.
- The **router service** provides the in-browsing navigation capabilities.



### 3.1. Angular application Architecture (3/3)



Modules



# #4 TypeScript

## 4. TypeScript ( 1/15)

- ❑ Angular is written in TypeScript.
- ❑ In order to build applications with Angular you need to be comfortable with TypeScript.
- ❑ For learning Angular, you need to understand the fundamentals of TypeScript and object programming principles. If need further documentation and examples, refer to the TypeScript's official handbook: <https://www.typescriptlang.org/assets/typescript-handbook-beta.pdf> .
- ❑ This section should provide you a good understanding of :
  - Type annotations,
  - Arrow functions,
  - Interfaces,
  - Classes,
  - Constructors,
  - Access modifiers,
  - Properties
  - Modules.

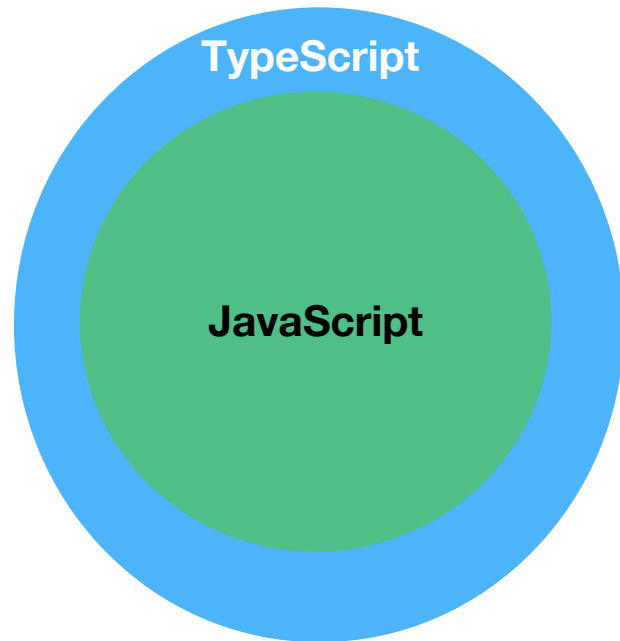
## 4. TypeScript (2/15)

### ❑ What is TypeScript?

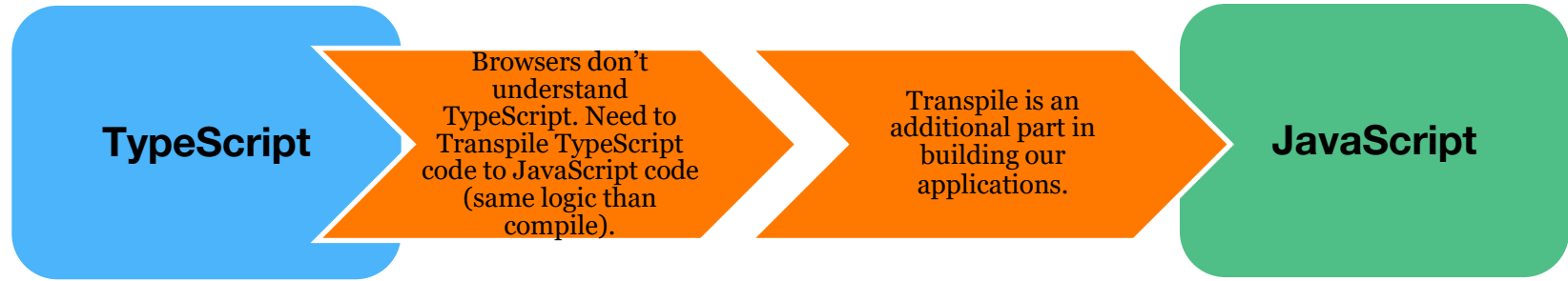
- It's not an entirely new language but it's just a superset of JavaScript
- Any valid JavaScript code is also a valid TypeScript code.
- TypeScript has additional features that do not exist in the current version of JavaScript supported by most of all browsers.

### ❑ TypeScript brings:

- **Strong/Static typing**. In C# or Java, when a variable is defined, its type needs to be specified. In TypeScript, **typing** is optional but using it makes the application more predictable and easier for debugging.
- Quite of **Object-oriented features** missed for a long time in JavaScript's earliest versions like : Classes, Interfaces, Constructors, access modifiers (like public and private), filled properties, generics, etc
- **Compile-time catching errors** instead of at runtime (TypeScript Transpiler see next slide).
- **Access to great tooling** like intelligence given in the code editors



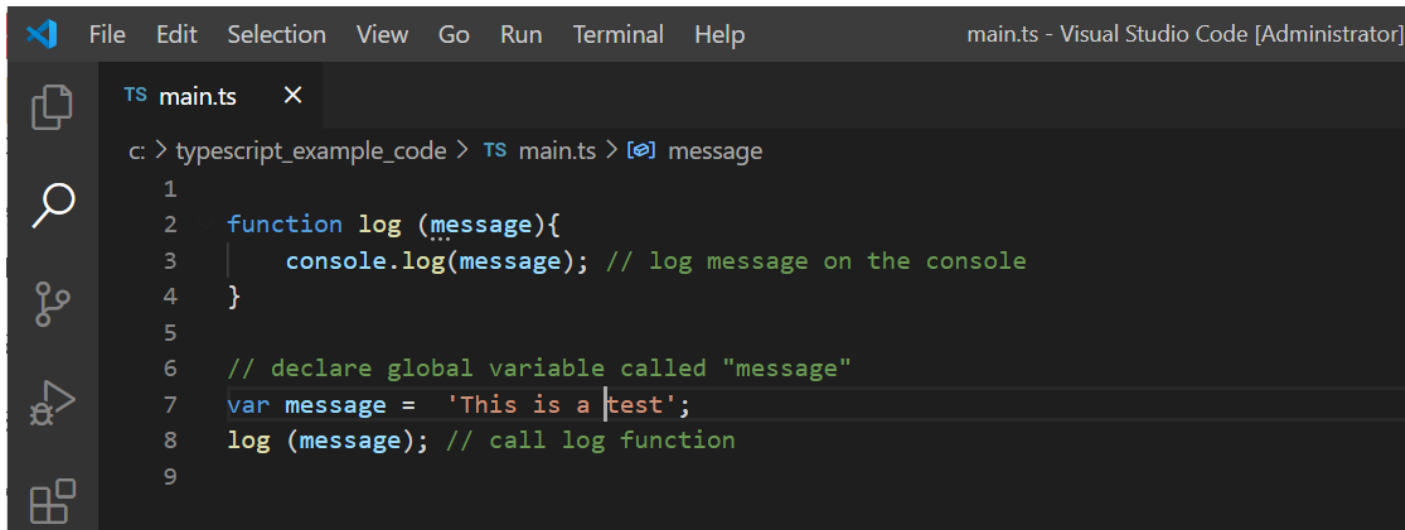
## 4. TypeScript ( 3/15)



- ❑ Install TypeScript if need to work independently than Angular's already installed environment : tap in window terminal **npm install --g typescript** (sudo at the front for Mac users). **tsc --version** to know the installed version of TypeScript compiler.

```
c:\>mkdir typescript_example_code // create a workspace folder for your TypeScript Code sample
c:\>cd typescript_example_code
c:\typescript_example_code>code // create a main.ts file and open it with Visual Studio Code
```

## 4. TypeScript ( 4/15)



```
File Edit Selection View Go Run Terminal Help main.ts - Visual Studio Code [Administrator]

TS main.ts X
c: > typescript_example_code > TS main.ts > [?] message
1
2 function log (message){
3     console.log(message); // log message on the console
4 }
5
6 // declare global variable called "message"
7 var message = 'This is a test';
8 log (message); // call log function
9
```

```
c:\typescript_example_code>tsc main.ts // Call TypeScript Transpiler to transpile your main.ts to main.js
```

```
c:\typescript_example_code>dir // or ls if you are working on Mac or Linux
```

```
20/12/2020 09:59      199 main.js
```

```
20/12/2020 09:52      205 main.ts
```

When building an Angular app, this transpilation or a compilation step happens under the hood. We don't have to manually call the TypeScript compiler. In fact, when we run the application using **ng serve**, Angular CLI calls TypeScript compiler which seamlessly transpiles all the related TypeScript code.



## 4. TypeScript ( 5/15)

```
c:\typescript_example_code>code main.js
// Open the main.js and let compare it to
main.ts
```

The main.js has exactly the same code than the one we wrote in main.ts → Thus, all valid JavaScript code is also a valid TypeScript code.

// To execute the main.js code in the terminal we use :

```
c:\typescript_example_code>node main.js
This is a test
```

```
TS main.ts JS main.js X
c: > typescript_example_code > JS main.js > log
1 function log(message) {
2     console.log(message); // log message on the console
3 }
4 // declare global variable called "message"
5 var message = 'This is a test';
6 log(message); // call log function
```

vs

```
TS main.ts X JS main.js
c: > typescript_example_code > TS main.ts > message
1
2 function log (message){
3     console.log(message); // log message on the console
4 }
5
6 // declare global variable called "message"
7 var message = 'This is a test';
8 log (message); // call log function
```

Orange Restricted

## 4. TypeScript ( 6/15)

### 4.1. Variable declarations

In JavaScript a variable declared with a “**var**” keyword is scoped to the nearest function where it is declared.

However, if we replace “var” by a “**let**” key word, we immediately got error from the code editor on this variable when it's called outside the 'for block' where it was initialized.

→ Mentioning errors before compilation is one of the helpful features that TypeScript brings

```
c: > typescript_example_code > TS main.ts > makesomething
1
2 function makesomething(){
3     for (let i=0; i<5; i++){
4         console.log(i);
5     }
6
7     console.log('Finally: ' + i);
8 }
9
10 makesomething();
```

[ts] Cannot find name 'i'.  
any

```
TS main.ts X
c: > typescript_example_code > TS main.ts > ...
1
2 function makesomething(){
3     for (var i=0; i<5; i++){
4         console.log(i);
5     }
6
7     console.log('Finally: ' + i);
8 }
9
10 makesomething();
11
```

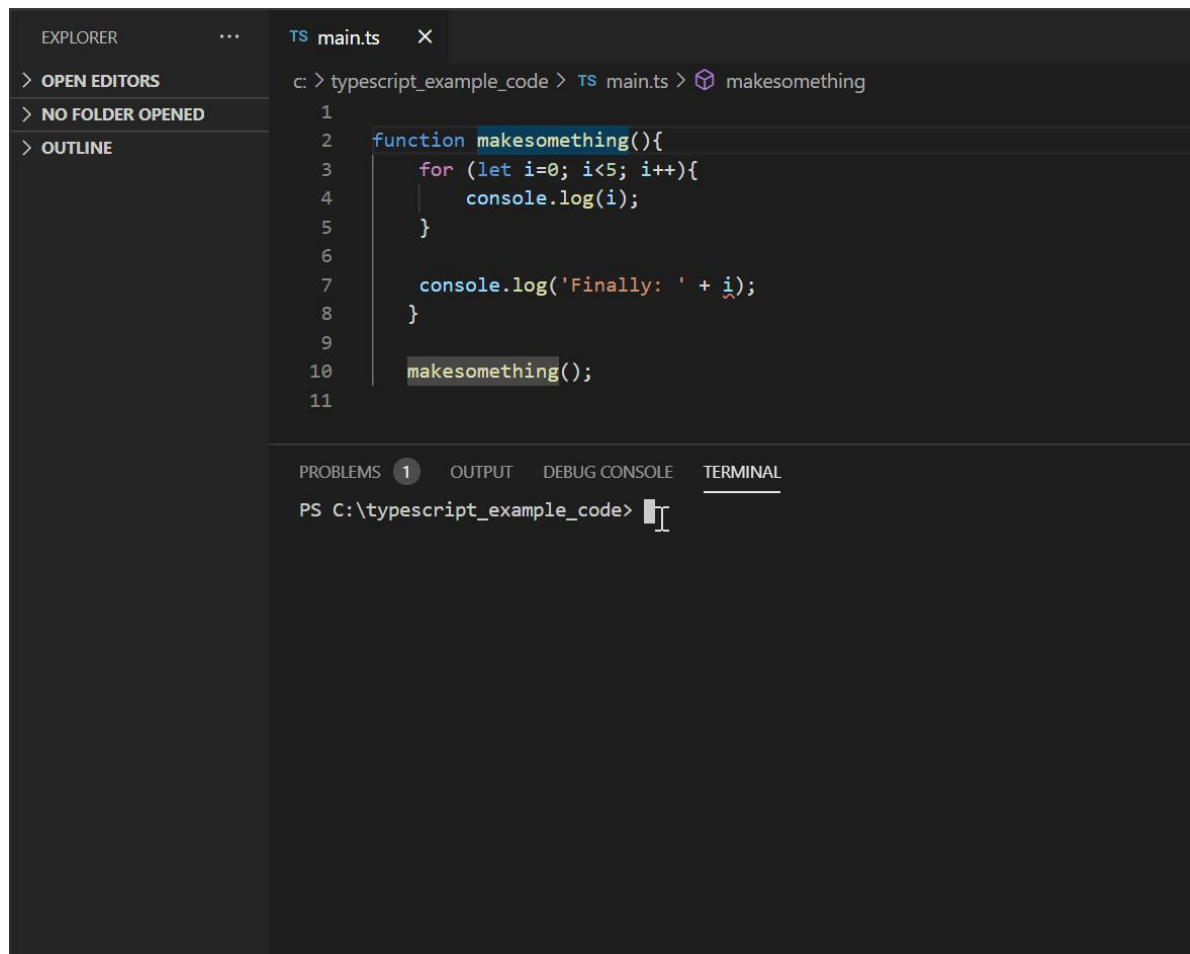
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\typescript_example_code> tsc main.ts | node main.js
0
1
2
3
4
Finally: 5
PS C:\typescript_example_code>
```

## 4. TypeScript ( 7/15)

### 4.1. Variable declarations

- ❑ Even if an error is reported, TypeScript compiler continue to generate “main.js”
- ❑ By default TypeScript compiler compile the code to JavaScript ECMA 5 (EC5) which is supported by the most known browsers.



```
EXPLORER  ...  TS main.ts  X
> OPEN EDITORS
> NO FOLDER OPENED
> OUTLINE

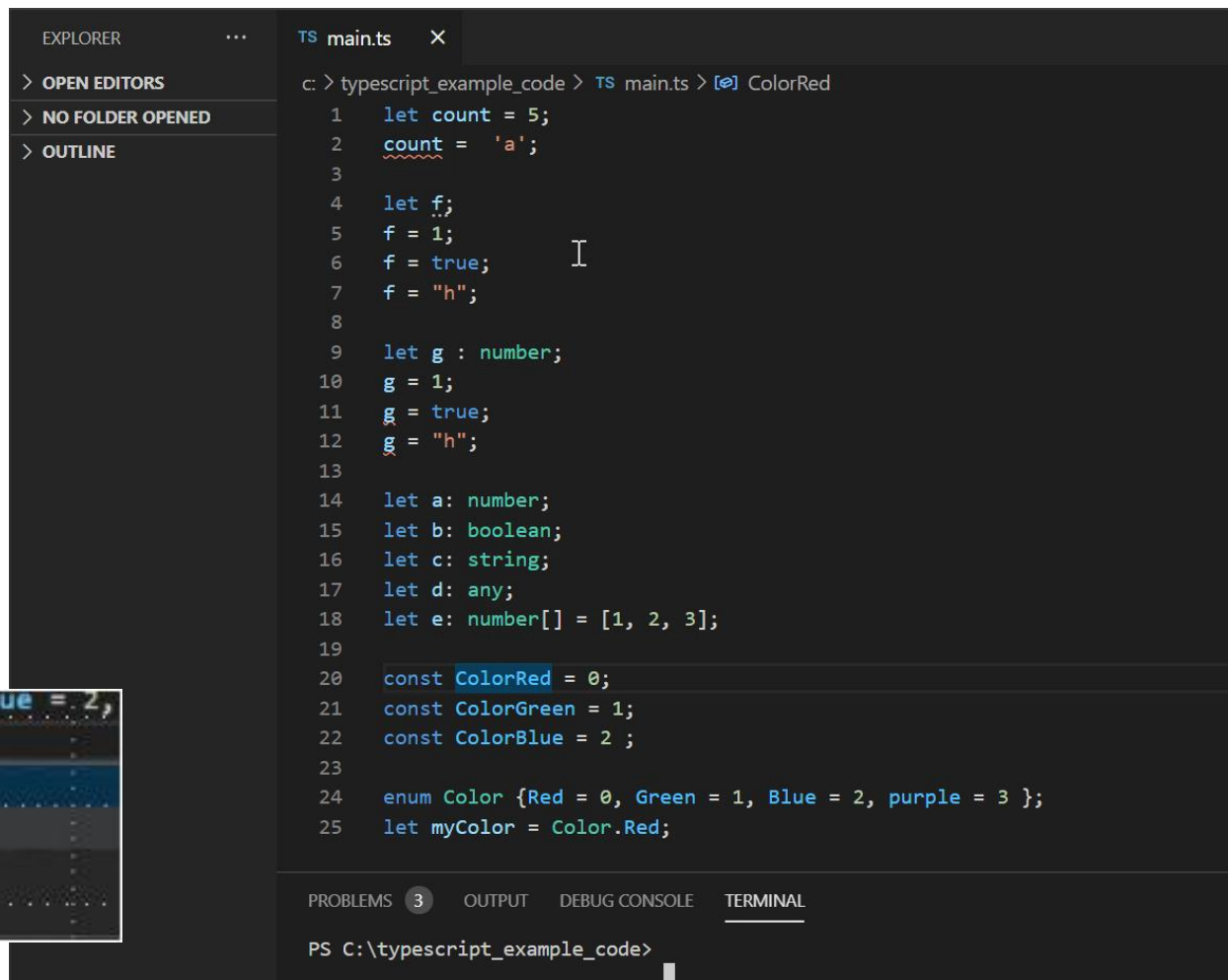
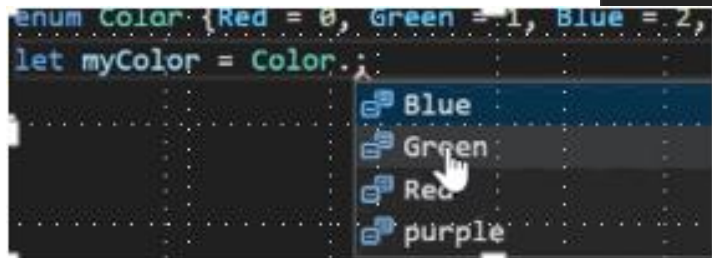
c: > typescript_example_code > TS main.ts > makesomething
1
2  function makesomething(){
3      for (let i=0; i<5; i++){
4          console.log(i);
5      }
6
7      console.log('Finally: ' + i);
8  }
9
10  makesomething();
11

PROBLEMS 1  OUTPUT  DEBUG CONSOLE  TERMINAL
PS C:\typescript_example_code>
```

## 4. TypeScript ( 8/15)

### 4.2.Different types in TypeScript

- ❑ In JavaScript we can change the type of variable, whereas it's not the case in TypeScript.
- ❑ Type annotations (number, Boolean, string, any, array number [1,2,3], array any [1, true, 'a']).
- ❑ In TypeScript like as in different oriented objects language 'enum' let us put a group of constants in a one container.



## 4. TypeScript ( 9/15)

### 4.3.Type Assertion

```
TS main.ts  X  JS main.js
c: > typescript example_code > TS main.ts > ...
1   let password: string
2   let password = 'lmno';
3   let endsWithO = password.endsWith ('o');
4
5   let userName;
6   userName = 'xyz';
7   let endsWithZ = userName.endsWith ('c');
8
9   let message;
10  message = 'abc';
11
12  // <string> is a way to make Type Assertion. Is the most used.
13  // It tells to TS compiler that "message" is a string.
14  let endsWithC = (<string>message).endsWith('c');
15
16  // A second way to make type assertion is by adding "as sting" after the variable message.
17  let alternativeWay = (message as string).endsWith('c');
18  /*
19  =====
20  = Type assertions are a way to tell the compiler "trust me, I know what I'm doing." =
21  = Type assertion does not change the type of the variable at runtime.             =
22  = It performs no special checking or restructuring of data in memory.             =
23  = It's purely a way to tell TypeScript compiler about the type of a variable       =
24  = to allow the developer access to the intelisense.                             =
25  =====
26  */
```

## 4. TypeScript ( 10/15)

### 4.4.Arrow functions

```
TS main.ts  JS main.js
c: > typescript_example_code > TS main.ts > ...
1
2
3   let log = function (value) {
4   |     console.log (value);
5   |   }
6
7   let doLog = (value) => {
8   |     console.log(value);
9   |   }
10
11   // if the function has only one line
12   // we can exclude the curly braces
13
14   let ifOneLine = (value) => console.log(value);
15
16   // if the function has now parameter we get this expression
17   let ifNoParameter = () => console.log ();
18
19
20   // Arrow function is called in C-Sharp Lambda expression
```

## 4. TypeScript ( 11/15)

### 4.5.Interfaces

- ❑ Can deal with a situation where a function has to received a lot of number of parameters
- ❑ Interface has the same concept than in C# or Java.
- ❑ With my Interface we defined the shape of an object.
- ❑ Interface are purely for declarations, so can't include inside of it implementations (algorithms or functions like “draw point, calculate distance, etc.).

### 4.6. Classes

- ❑ Class comes to answer the principle of **Cohesion** in object-oriented programming language.
- ❑ **Cohesion concept** means, things that are related should be part of one unit, they should go together.
- ❑ A Class groups variables (properties) and functions (methods) that are highly related.

```
interface Point {  
  x: number,  
  y: number  
}
```

```
let drawPoint = (point: Point) => {  
  //...  
}
```

```
class Point {  
  x: number; //fields  
  y: number; //fields  
  //methods  
  draw(){  
    console.log('X: ' + this.x + 'Y: ' + this.y);  
  }  
  getDistance(another: Point){  
    //...  
  }  
}
```

## 4. TypeScript ( 12/15)

### 4.7. Objects

- ❑ An object is instance of a Class.

### 4.8. Constructors

- ❑ Every Class has a Constructor which is basically a method that is called when we create an instance of that class.

### 4.9. Access modifiers

- ❑ Key words that we can apply to a member of a class to control its access from outside this class.
- ❑ In TypeScript we have three access modifiers : **public**, **private** and **protected**.
- ❑ By default all members are public if not specified
- ❑ Access modifiers can be applied on field, properties and methods.

```
class Point {  
    x: number;  
    private y: number;  
  
    constructor (x?: number, y?: number){  
        this.x = x;  
        this.y = y;  
    }  
  
    draw(){  
        console.log('X: ' + this.x + 'Y: ' + this.y);  
    }  
}  
  
let point = new Point();  
point.draw();
```



## 4. TypeScript ( 13/15)

### 4.10. Access modifiers in constructor parameters

- ❑ If a constructor parameter is prefixed with an access modifier (private or public), TypeScript Compiler will generate a field with exactly the same name and would also initialize this field with a value of its argument. So this feature is very helpful to get a less verbose code.

```
class Point {  
    /* private x: number;  
    private y: number; */  
  
    //we deleted these two fields and in place we prefixed  
    //our parameters with access modifiers in the constructor  
    constructor (private x?: number, private y?: number){  
        /* this.x = x;  
        this.y = y;*/  
        //then also we don't need these repetitive assignments  
    }  
  
    draw(){  
        console.log('X: ' + this.x + 'Y: ' + this.y);  
    }  
}  
  
let point = new Point();  
point.draw();
```

## 4. TypeScript ( 14/15)

### 4.11. Properties

- ❑ By setting in the previous slide the private access modifier, we can initialize the coordinates of our point and also can draw it but we have no longer a way to read its two coordinates outside our class. To solve this issue we can use feature called **Property**.
- ❑ **Property** looks like a field from the outside of the class but internally of that class it's really a method.
- ❑ Property it's either one method which is a getter or setter or a combination of a getter and a setter. We can

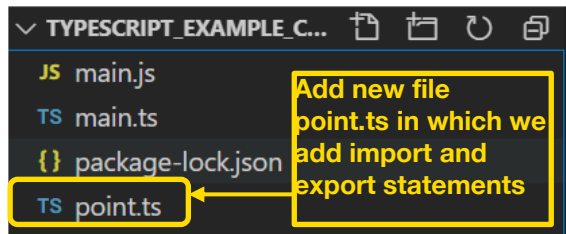
```
class Point {
    constructor (private _x?: number, private _y?: number){
    }
    draw(){
        console.log('X: ' + this._x + 'Y: ' + this._y);
    }
    get x(){
        return this._x;
    }
    set x(value){
        if (value<0)
            throw new Error ('value cannot be less than 0.')
        this._x = value;
    }
}

let point = new Point();
let x=point.x;
point.x = 10;
point.draw();
```

## 4. TypeScript ( 15/15)

### 4.12. Modules

- ❑ In TypeScript we divide our program in a multiple files. In each file we export one or more types which can be functions, classes, simple variables or objects. Wherever we need to use them we have to import them first.
- ❑ When we have an import or export statement in top of a file, this one is then considered as Module from TypeScript point of view.



```
//main.ts
import {Point} from './point'
let point = new Point();
point.draw();
```

```
//initially we have our class in a main.ts
class Point {
    constructor (private x?: number, private y?: number){
    }
    draw(){
        console.log('X: ' + this.x + 'Y: ' + this.y);
    }
}
let point = new Point();
let x=point.x;
point.x = 10;
point.draw();
```

```
export class Point {

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

    }

    draw(){
        console.log('X: ' + this.x + 'Y: ' + this.y);
    }

}
```

# #5

# Angular's building blocks

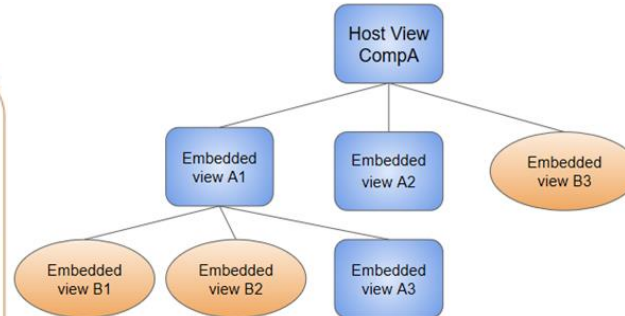
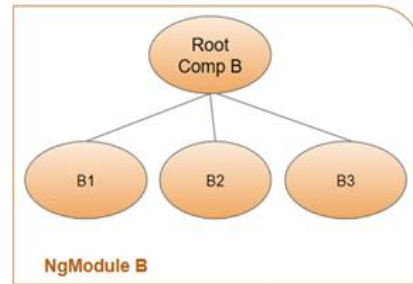
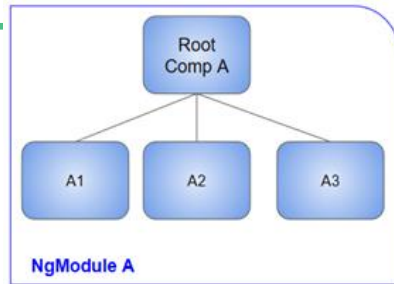
## 5.1. Modules

- ❑ Modules in Angular are called NgModules.
- ❑ NgModules are containers for a cohesive block of code dedicated to an app domain, a workflow, or a closely related set of capabilities.
- ❑ They contain components, service providers, and other block of code files.
- ❑ Import & export functionalities between #NgModules
- ❑ Every Angular app has at least one NgModule class (root module) named AppModule. Resides in a file named app.module.ts.
- ❑ The angular app is launched by bootstrapping the root NgModule.
- ❑ NgModules provide a compilation context for their components.

```
//src/app/app.module.ts == > NgModule example
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
@NgModule({
  imports: [ BrowserModule ],
  providers: [ Logger ],
  declarations: [ AppComponent ],
  exports: [ AppComponent ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

A decorator function

Properties



hierarchical structure of views is a key factor in the way Angular detects and responds to changes in the DOM and app data

## 5.2. Components

- ❑ Component controls the views on the screen.
- ❑ Component is a class that defines the application logic which support our view through methods and properties.
- ❑ Angular creates, updates, and destroys components as the user moves through the application. Your app can take action at each moment in this lifecycle through optional **lifecycle hooks**, like `ngOnInit()`.

- ❑ **Selector** : a CSS selector that tells Angular to create and insert an instance of this component wherever it finds the corresponding tag (ex. `<app-hero-list></app-hero-list>` ) in the HTML template.
- ❑ **templateUrl**: the module-relative address of this component's HTML template.
- ❑ **providers**: an array of providers for services that the component requires

```
//src/app/hero-list.component.ts (class)
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-hero-list',
  templateUrl: './hero-list.component.html',
  providers: [ HeroService ]
})

export class HeroListComponent implements OnInit {
  heroes: Hero[];
  selectedHero: Hero;

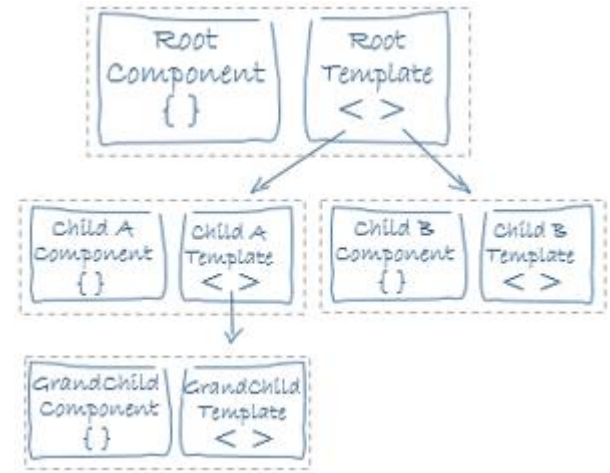
  constructor(private service: HeroService) { }

  ngOnInit() {
    this.heroes = this.service.getHeroes();
  }

  selectHero(hero: Hero) { this.selectedHero = hero; }
}
```

## 5.3. Templates

- ❑ A template is a form of HTML that tells Angular how to render the component.
- ❑ Views (Component + template) are arranged hierarchically allowing to display pages as unit.
- ❑ The template immediately associated with a component defines that **component's host view**.
- ❑ The component can also define a view hierarchy, which contains embedded views, hosted by other components.
- ❑ A view hierarchy can include views from components in the same NgModule, but can (and often does) include views from components defined in others NgModules



- ❑ **\*ngFor is a directive** that tells Angular how to iterate over a list.
- ❑ The syntaxes **{{hero.name}}**, **(click)**, and **[hero]** bind program data to and from the DOM, responding to user input.
- ❑ **providers**: an array of providers for services that the component requires.
- ❑ The tag **<app-hero-detail>** is an element that represents a new component, HeroDetailComponent (code not shown here, defines the hero-detail child view of HeroListComponent).

```
//src/app/hero-list.component.html
```

```
<h2>Hero List</h2>
```

```
<p><i>Pick a hero from the list</i></p>
```

```
<ul>
```

```
  <li *ngFor="let hero of heroes" (click)="selectHero(hero)">
    {{hero.name}}
```

```
  </li>
```

```
</ul>
```

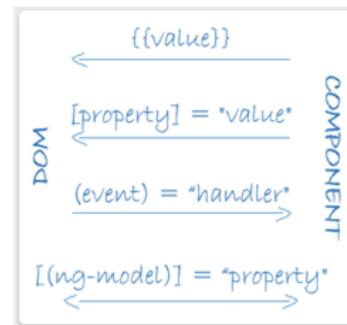
```
<app-hero-
```

```
detail *ngIf="selectedHero" [hero]="selectedHero"></app-hero-
detail>
```

Orange Restricted

## 5.4. Data binding (1/2)

- ❑ **Data binding** eases pushing data values into the HTML page, controls and turn user responses into actions and value updates.
- ❑ Data binding is a mechanism for coordinating the parts of a template with the parts of a component.
- ❑ Two-way data binding are supported in Angular.
- ❑ Four markup forms of data binding (direction: to the DOM, from the DOM, or both).



```
//src/app/hero-list.component.html (binding)
```

```
<li>{{hero.name}}</li>
```

**{{hero.name}}** is an **interpolation**: displays the component's hero.name property value within the <li> element.

```
<app-hero-detail [hero]="selectedHero"></app-hero-detail>
```

```
<li (click)="selectHero(hero)"></li>
```

**[hero]** is a **property binding**: passes the value of selectedHero from the parent HeroListComponent to the hero property of the child HeroDetailComponent

**(click)** is an **event binding** calls the component's selectHero method when the user clicks a hero's name.

```
//src/app/hero-detail.component.html (ngModel)
```

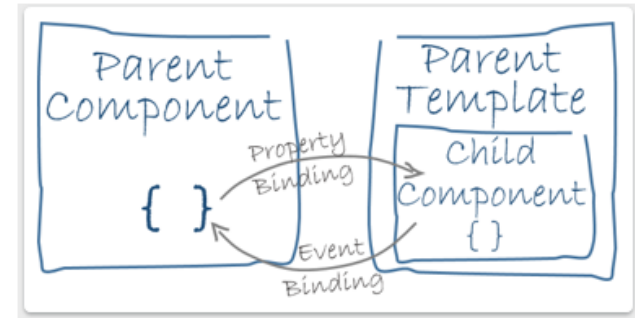
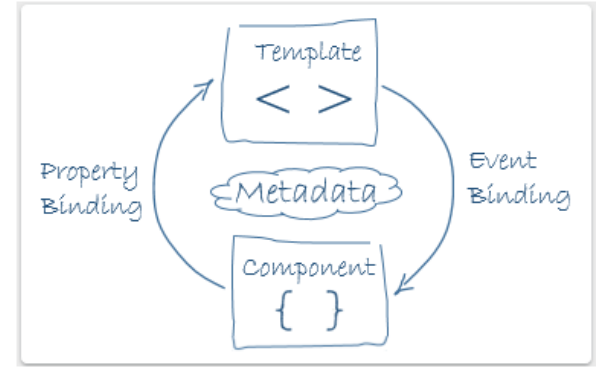
```
<input [(ngModel)]="hero.name">
```

Two-way data binding with the **ngModel directive** combines property and event binding in a single notation.



## 5.4. Data binding (2/2)

- ❑ Two-way data binding combines property and event binding in a single notation. Is mainly used in **template-driven forms**.
- ❑ In two-way binding, a data property value flows to the input box from the component as with property binding. The user's changes also flow back to the component, resetting the property to the latest value, as with event binding
- ❑ Angular processes all data bindings once for each JavaScript event cycle, from the root of the application component tree through all child components.
- ❑ Data binding plays an important role in communication between a template and its component, and is also important for communication between parent and child components.



## 5.5. Pipes

- ❑ Pipes let declare display-value transformations in HTML template HTML.
- ❑ Various pipes are defined in Angular such as: date pipe, currency pipe, etc. (<https://angular.io/api?type=pipe>).
- ❑ Pipe operator “|” → {{interpolated\_value | pipe\_name}}. It's used to specify value transformation in HTML template.
- ❑ Pipes can be chained. Output of one pipe can be injected in input to another pipe function.
- ❑ A class with the **@Pipe decorator** defines a function that transforms input values to output values for display in a view

```
<!-- Default format: output 'Jun 15, 2015'-->
<p>Today is {{today | date}}</p>

<!-- fullDate format: output 'Monday, June 15, 2015'-->
<p>The date is {{today | date:'fullDate'}}</p>

<!-- shortTime format: output '9:43 AM'-->
<p>The time is {{today | date:'shortTime'}}</p>
```

```
//Example of Transforming Data by Using Pipes
//src/app/power-boost-calculator.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-power-boost-calculator',
  template: `
    <h2>Power Boost Calculator</h2>
    <div>Normal power: <input [(ngModel)]="power"></div>
    <div>Boost factor: <input [(ngModel)]="factor"></div>
    <p>
      Super Hero Power: {{power | exponentialStrength: factor}}
    </p>
  `
})
export class PowerBoostCalculatorComponent {
  power = 5;
  factor = 1;
}
```

## 5.6. Directives

- ❑ A directive is a class with a `@Directive()` decorator.
- ❑ As Angular templates are dynamic, when they are rendered they transform the DOM according to the instructions given by the directives.
- ❑ A component is technically a directive. However, components are so distinctive and central to Angular applications that Angular defines the `@Component()` decorator, which extends the `@Directive()` decorator with template-oriented features.
- ❑ The metadata for a directive associates the decorated class with the selector element used to insert it into HTML.
- ❑ Directives appear as attributes in the template, either by name or as the target of an assignment or a binding.
- ❑ → 3 kinds of directives : **components without view features**, **structural** and **attribute**.
- ❑ Pre-defined directives: alter the layout structure (ex. `ngSwitch`), or modify DOM and components (ex. `ngStyle` and `ngClass`).

**Structural directives** alter layout by adding, removing, and replacing elements in the DOM.

- `*ngFor` is an iterative; it tells Angular to stamp out one `<li>` per hero in the heroes list.
- `*ngIf` is a conditional; it includes the `HeroDetail` component only if a selected hero exists

```
//src/app/hero-list.component.html (structural)
<li *ngFor="let hero of heroes"></li>
<app-hero-detail *ngIf="selectedHero"></app-hero-detail>
```

**Attribute directive** alter the appearance or behavior of an existing element. In templates they look like regular HTML attributes, hence the name.

- The `ngModel` directive (which implements two-way data binding) is an example of an attribute directive. It modifies the behavior of an existing element (typically `<input>`) by setting its display value property and responding to change events.

```
//src/app/hero-detail.component.html (ngModel)
<input [(ngModel)]="hero.name">
```

## 5.7. Services and Dependency Injection (DI)

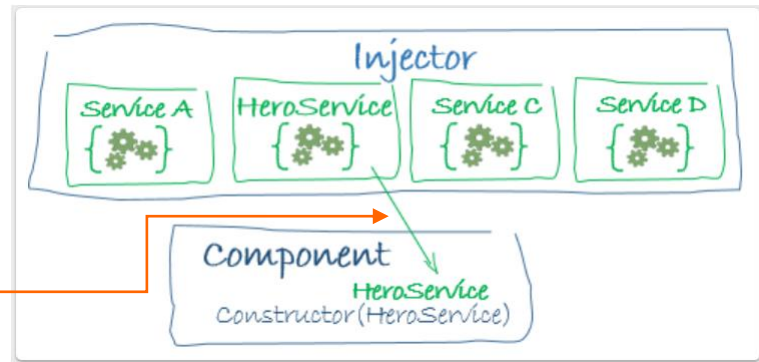
- ❑ A Service is a broad category encompassing any value, function, or feature that an app needs.
- ❑ A service is typically a class with a narrow, well-defined purpose. It should do something specific and do it well.
- ❑ Services examples: fetching data from the server, validating user input, or logging directly to the console.
- ❑ The services are made available to any component with an **injectable service class**.

- ❑ **Dependency Injection** is wired into the Angular framework and used everywhere to provide new components with the services or other things they need.
- ❑ Services are made to be injected to any component that needs that service.
- ❑ A class is defined as serviced by using the decorator function `@Injectable`.
- ❑ A provider is an object that tells an injector how to obtain or create a dependency.
- ❑ When all requested services have been resolved and returned, Angular can call the component's constructor with those services as arguments

```
//src/app/hero-detail.component.html (ngModel)
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable({
  providedIn: 'root'
})

export class CartService {
  //...
}
```



# #6

## Exercise

## Make your Angular app

# References

- ❑ Angular official web page: <https://angular.io/>



# Thanks !

# Questions ?

Samir Bellahsene

[samir.bellahsene@orange.com](mailto:samir.bellahsene@orange.com)