Angular Fundamentals

.NET CORE

Angular is an application design framework and development platform for creating efficient and sophisticated single-page apps.

[HTTPS://ANGULAR.IO/DOCS](https://angular.io/docs)

TS/Angular Workspace SetUp

https://angular.io/guide/setup-local
https://code.visualstudio.com/docs/typescript/typescript-compiling
https://code.visualstudio.com/docs/typescript/typescript-compiling
https://angular.io/tutorial/toh-pt0#create-a-new-workspace-and-an-initial-application

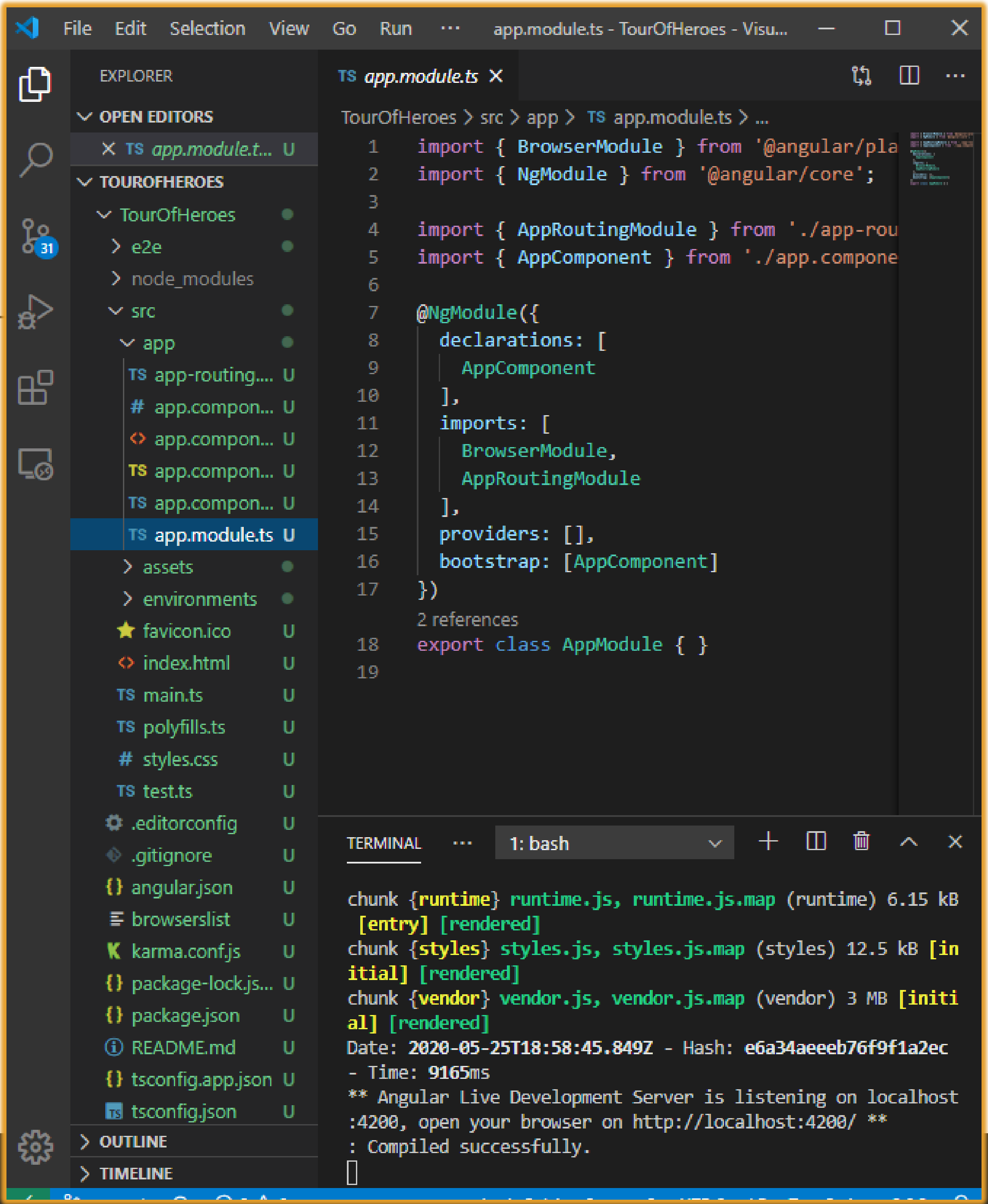
Following the steps from here to create your first Angular App.

1. Make sure you have Node.js with node –v in Command Line. If not, go to nodejs.org to get it.
2. Install Angular CLI globally with npm install -g @angular/cli in Command Line.
3. Use ng new my-app-name to create a *WorkSpace* for your app and install the default starter app.
4. Press enter to accept the defaults.
5. ng new installs the Angular *npm* packages needed.
6. Navigate in the CLI to your app folder. (cd my-app-name).
7. Use ng serve –open to launch the server and open the browser with the default sample project.
8. In VS Code, install the *Angular Extension Pack* to get goodies!
9. Use this Angular Cheat Sheet for quick reference!

WorkSpace

https://angular.io/tutorial/toh-pt0#set-up-your-environment

A workspace contains all the files for one or more projects. A project is the set of files that comprise an app, a library, or end-to-end (e2e) tests.



```
import { BrowserModule } from '@angular/pla
import { NgModule } from '@angular/core';

import { AppRoutingModule } from './app-rou
import { AppComponent } from './app.compone

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

```
chunk {runtime} runtime.js, runtime.js.map (runtime) 6.15 kB
 [entry] [rendered]
chunk {styles} styles.js, styles.js.map (styles) 12.5 kB [in
itial] [rendered]
chunk {vendor} vendor.js, vendor.js.map (vendor) 3 MB [initi
al] [rendered]
Date: 2020-05-25T18:58:45.849Z - Hash: e6a34aeeeb76f9f1a2ec
- Time: 9165ms
** Angular Live Development Server is listening on localhost
:4200, open your browser on http://localhost:4200/ **
: Compiled successfully.
```

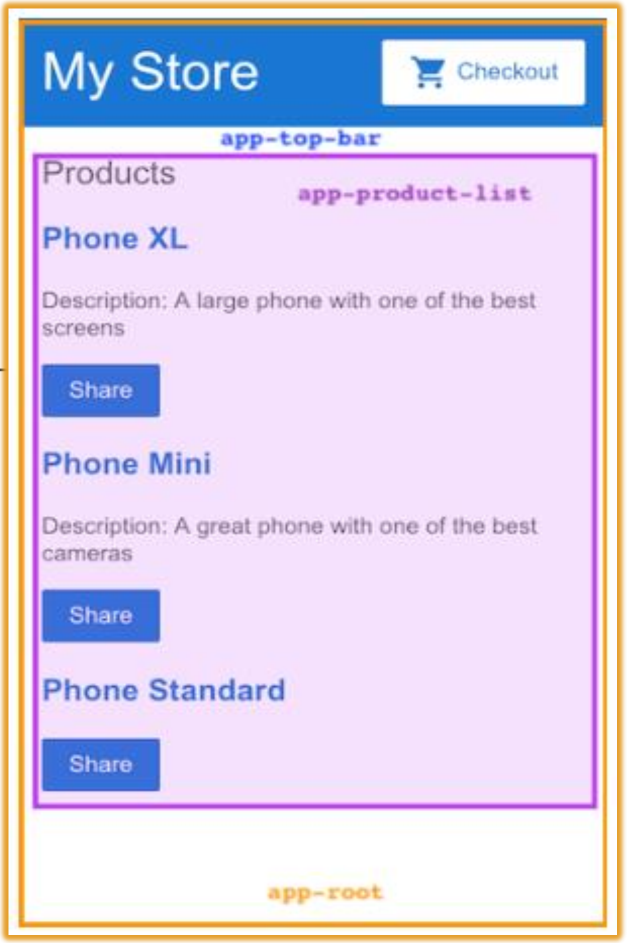
Components

https://angular.io/tutorial/toh-pt0#set-up-your-environment
https://angular.io/guide/component-interaction

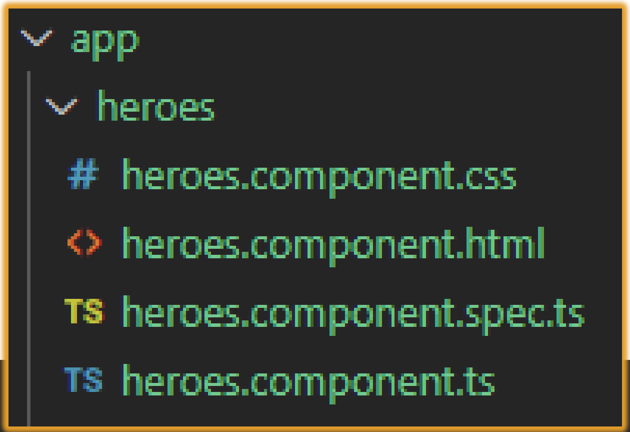
Components are the fundamental building blocks of ***Angular*** applications. They display data on the screen, listen for user input, and take action based on that input.

An ***Angular*** application comprises a tree of ***components***, in which each ***Angular component*** has a specific purpose and responsibility. In this example:

- app-root (orange box) is the application shell. This is the first component to load and the parent of all other components. You can think of it as the base page.
- app-top-bar (blue background) is the store name and checkout button.
- app-product-list (purple box) is the product list that you modified in the previous section.



My Store
Checkout
app-top-bar
Products
app-product-list
Phone XL
Description: A large phone with one of the best screens
Share
Phone Mini
Description: A great phone with one of the best cameras
Share
Phone Standard
Share
app-root



Angular Components

https://angular.io/tutorial/toh-pt1#create-the-heroes-component

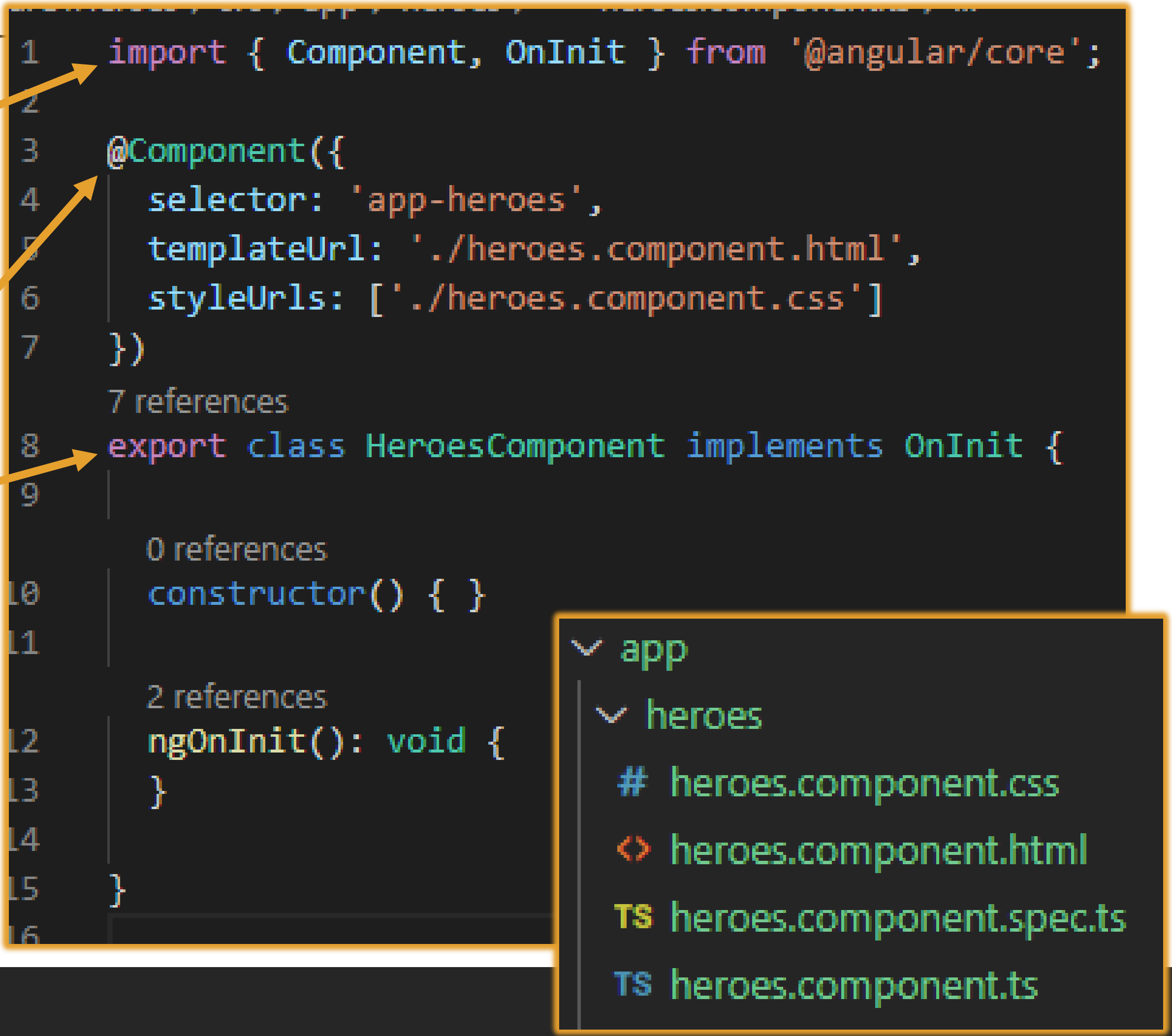
The *CLI* creates a new folder for each ***component*** and generates three files, .css, .ts, .html, inside it. User either the Angular helper (R-click the app folder) or the command ng generate component [name] to create a new ***component***.

When creating a Component, always import the ***Component symbol*** from the Angular core library and annotate the ***component class*** with @Component. @Component is a ***decorator*** function that specifies the Angular metadata for the ***component***:

- The selector name to use for CSS and if importing this component into a .html page.
- The relative .html location.
- The relative .css location.

The class uses the export keyword. This makes the class available to import by other components.

ngOnInit() is a lifecycle hook. It's a good place for component initialization logic like getting data from a Service.



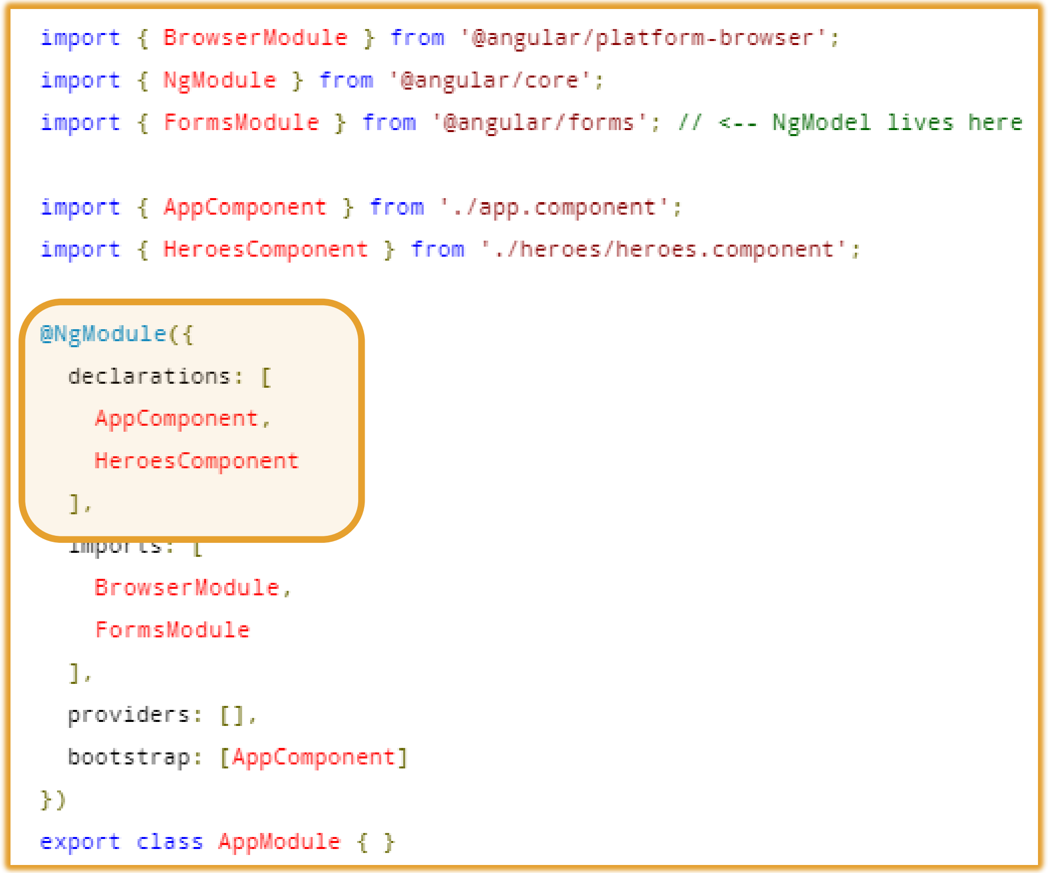
```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-heroes',
  templateUrl: './heroes.component.html',
  styleUrls: ['./heroes.component.css']
})
7 references
export class HeroesComponent implements OnInit {

  0 references
  constructor() { }

  2 references
  ngOnInit(): void {
  }

}
```

```
app
  heroes
    heroes.component.css
    heroes.component.html
    heroes.component.spec.ts
    heroes.component.ts
```

Connect a new Component

https://angular.io/tutorial/toh-pt1#show-the-heroescomponent-view

Every ***component*** must be declared in exactly one ***NgModule*** to function. When you declare a new ***component***, ***Angular CLI*** automatically imports the new component into ***app.module.ts*** and declares it under the @NgModule.declaration array on generation.



```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms'; // <-- NgModel lives here

import { AppComponent } from './app.component';
import { HeroesComponent } from './heroes/heroes.component';

@NgModule({
  declarations: [
    AppComponent,
    HeroesComponent
  ],
  imports: [
    BrowserModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

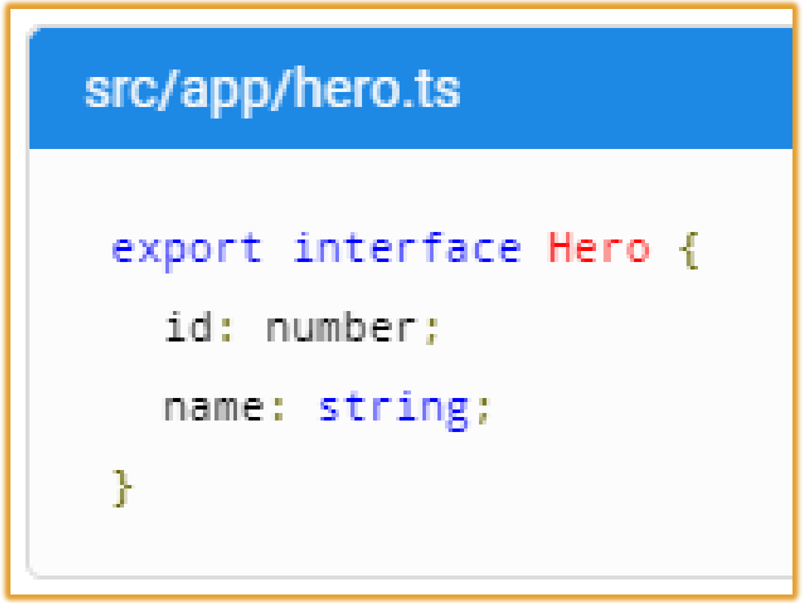
Create an interface

https://angular.io/tutorial/toh-pt1#create-a-hero-interface

Interfaces are useful for when you want to define a class or object, then implement it in various places.

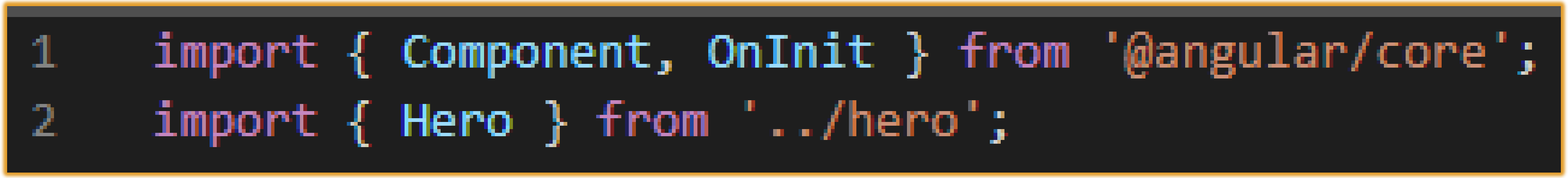
Create an *interface* with ng generate interface [name], or R-click the app folder => choose another schematic.

Then import that *interface* into the *Component* from the relative file location in which you want to use it.



src/app/hero.ts

```
export interface Hero {
  id: number;
  name: string;
}
```



```
import { Component, OnInit } from '@angular/core';
import { Hero } from '../hero';
```

TypeScript Modules

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

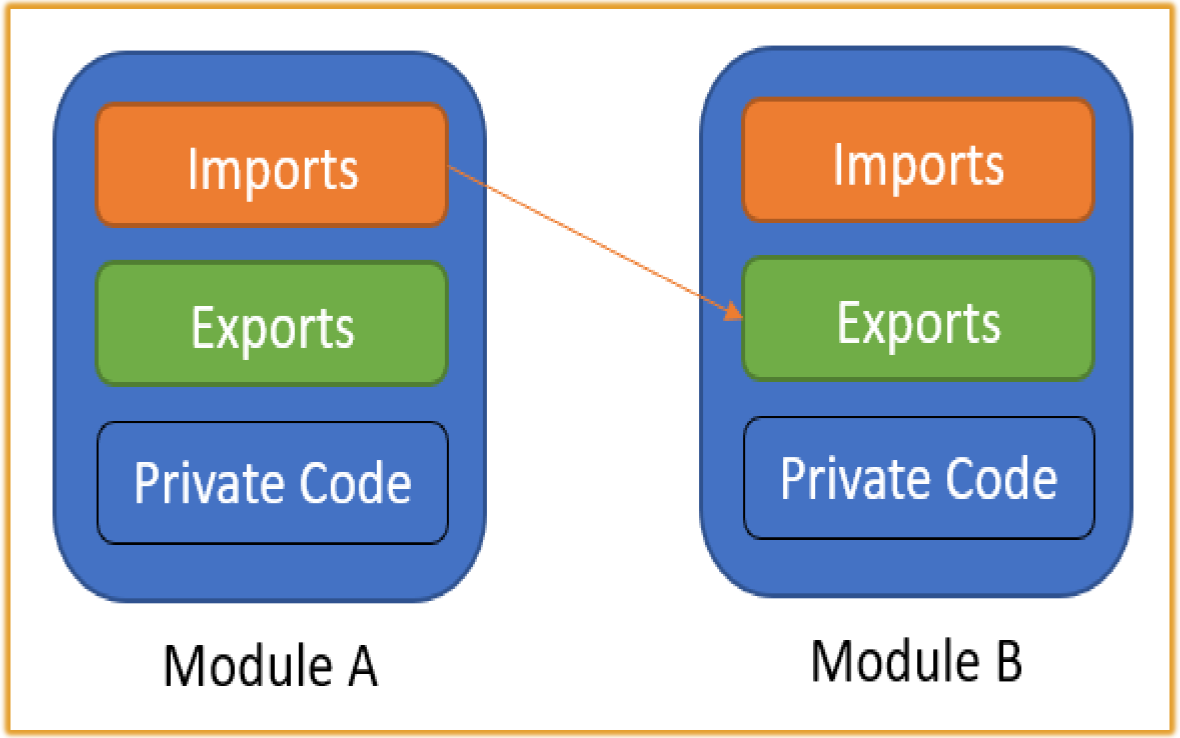
TS shares the *JS* concept of ***Modules***.

Modules in *TS* have their own scope. Anything declared inside a ***module*** is not visible outside that ***module*** unless it is explicitly ***exported***.

To consume a property ***exported*** from a different ***module***, it must be ***imported*** using an ***import*** method.

The relationships between ***modules*** are specified in terms of ***imports*** and ***exports*** at the file level.

In *TS*, any file containing a top-level ***import*** or ***export*** is considered a ***module***. A file without any top-level ***import*** or ***export*** declarations is treated as a script whose contents are available in the global scope (and therefore in ***modules*** as well).



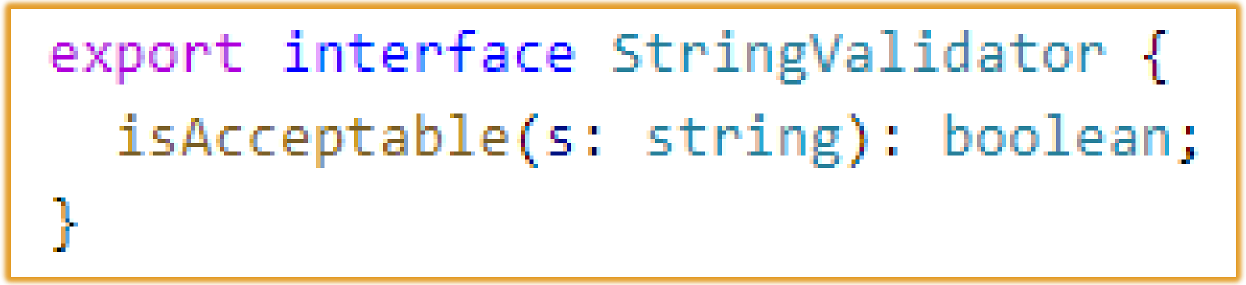
TypeScript - Exporting a Declaration

https://www.typescriptlang.org/docs/handbook/modules.html#export

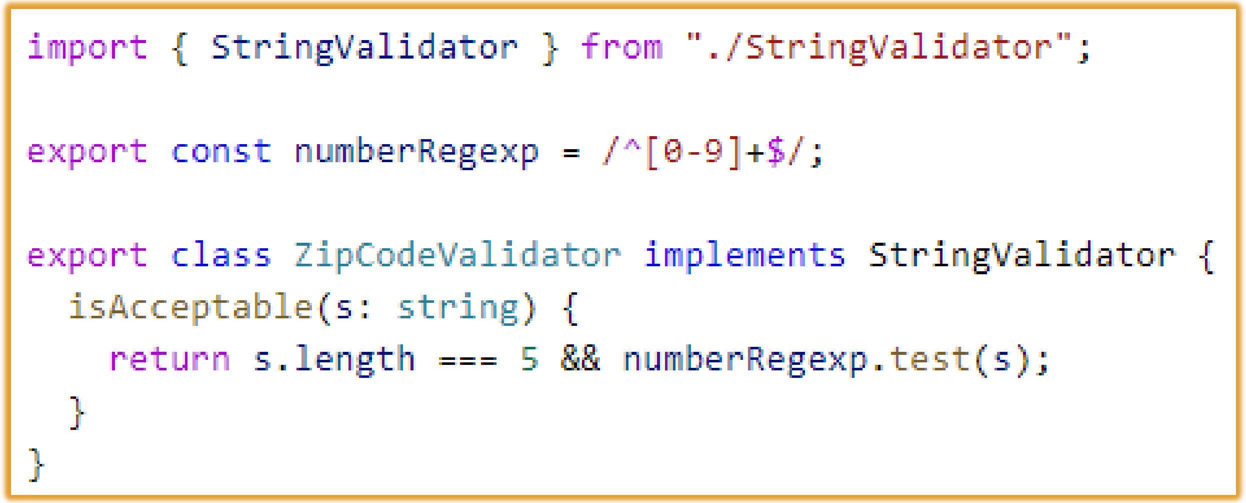
Any declaration (such as a variable, function, class, type alias, or interface) can be ***exported*** by adding the ***export*** keyword.

First, use the ***export*** keyword to make a class, function, or variable available to other ***modules*** from within the ***module*** (***component***).

Second, ***import*** the class, function, or variable into the ***module*** (***component***) where you want to implement it.



```
export interface StringValidator {
  isAcceptable(s: string): boolean;
}
```

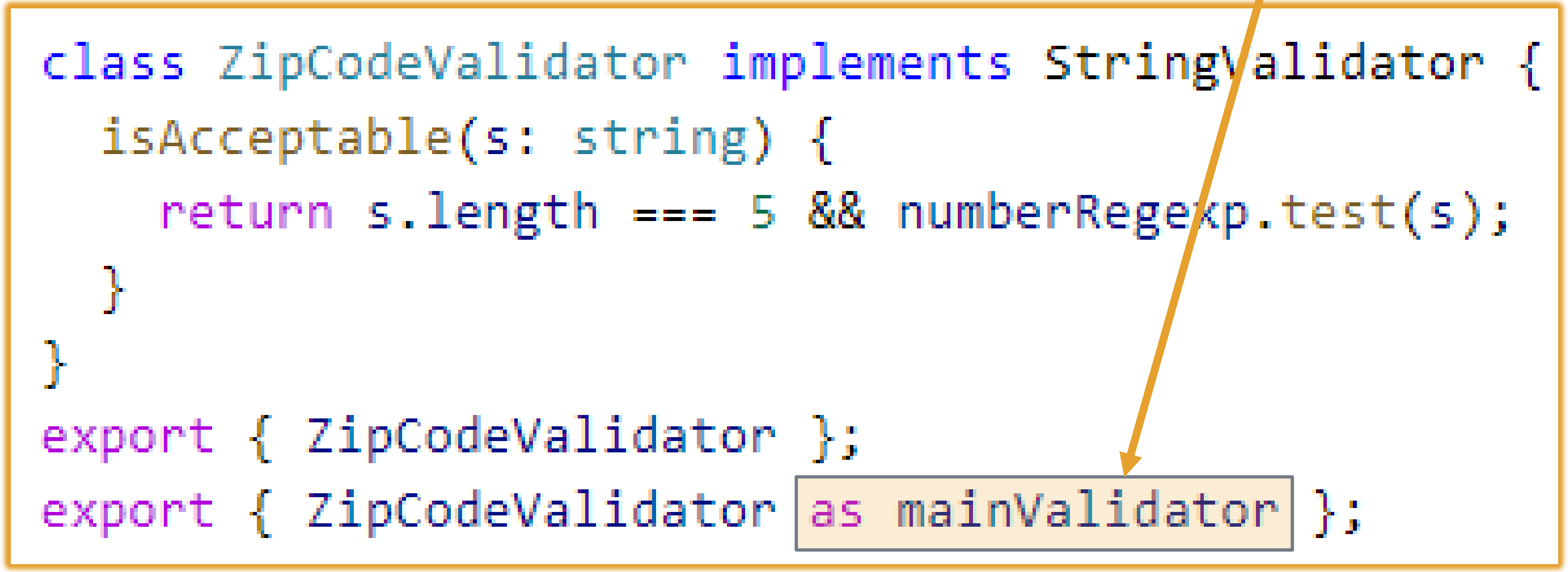


```
import { StringValidator } from "./StringValidator";

export const numberRegexp = /^[0-9]+$/;

export class ZipCodeValidator implements StringValidator {
  isAcceptable(s: string) {
    return s.length === 5 && numberRegexp.test(s);
  }
}
```

TypeScript - Export

https://www.typescriptlang.org/docs/handbook/modules.html#export-statements

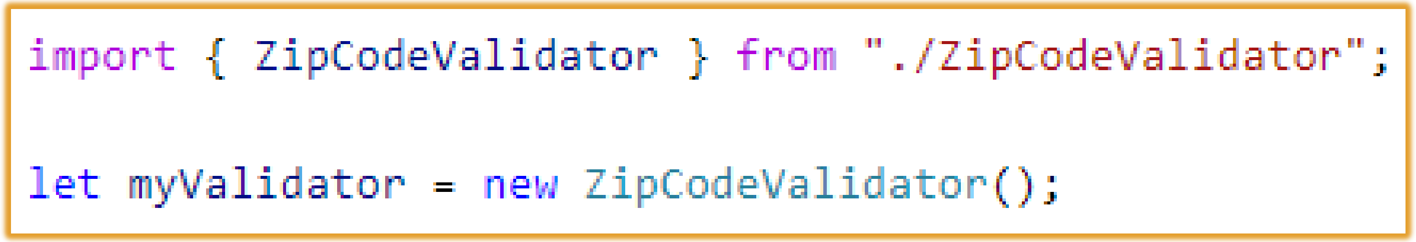
Export Statements allow you to rename the statement you want to export.



```
class ZipCodeValidator implements StringValidator {
  isAcceptable(s: string) {
    return s.length === 5 && numberRegexp.test(s);
  }
}
export { ZipCodeValidator };
export { ZipCodeValidator as mainValidator };
```

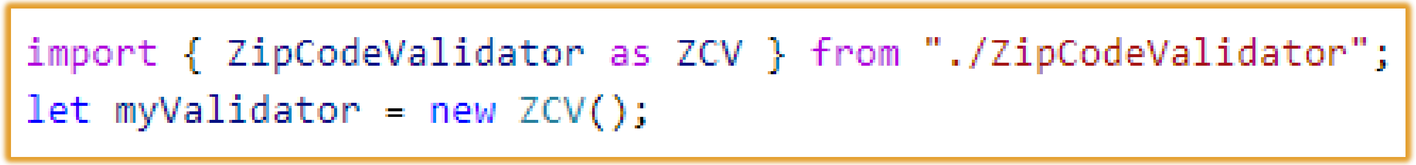
TypeScript Imports

https://www.typescriptlang.org/docs/handbook/modules.html#import



```
import { ZipCodeValidator } from "./ZipCodeValidator";

let myValidator = new ZipCodeValidator();
```

Imports can also be renamed.



```
import { ZipCodeValidator as ZCV } from "./ZipCodeValidator";
let myValidator = new ZCV();
```

Angular Templates - Data Binding

https://angular.io/tutorial/toh-pt1#two-way-binding

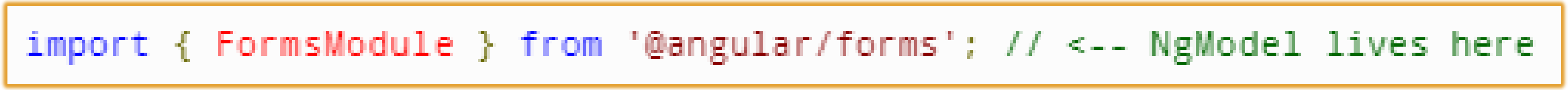
[(ngModel)] is Angular's two-way ***data binding*** syntax. It ***binds*** the property to the HTML so that data can flow in both directions: from the property to the textbox, and from the textbox back to the property.

ngModel isn't available by default do you have to import its module into the app. ***ngModel*** belongs to ***FormsModule*** and you have to opt-in (***import*** it) to use it.

@ngModule decorators have the needed metadata for the app to function.

The most important ***@NgModule decorator*** annotates the top-level ***AppModule*** class.

In app.module.ts, import the ***FormsModule***.



```
import { FormsModule } from '@angular/forms'; // <-- NgModel lives here
```

Then, add ***FormsModule*** to the imports array in the same file.

Modeling – Data Binding

https://angular.io/guide/template-syntax#property-binding
https://angular.io/tutorial/toh-pt3#update-the-heroescomponent-template

The double curly braces ({}) are ***Angular's*** interpolation binding syntax. This interpolation binding presents the component's property ***values*** inside the accompanying HTML Doc.

Property binding with [] around the property to be bound. This is one-way.

```
[class.selected]="hero === selectedHero"
```

Event binding based on events like 'click' or 'hover' to methods in the .ts file) using ().

```
<button (click)="addToCart(product)">Buy</button>
```

Two-Way Binding.

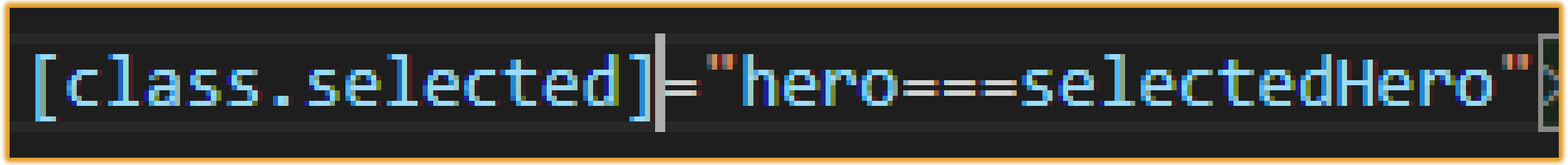
```
<input [(ngModel)]="hero.name" placeholder="name"/>
```

Angular Templates- Class Binding

https://angular.io/guide/template-syntax#class-binding

You can add and remove CSS class names from an element's class attribute with a ***class binding***.

To create a single ***class binding***, start with the prefix class followed by a dot (.) and the name of the ***CSS class*** ([class.foo]=“condition"). ***Angular*** adds the class when the bound expression is ***truthy***, and it removes the class when the expression is ***falsy***.

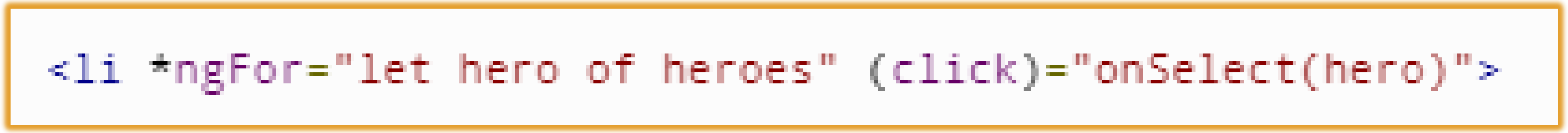


```
[class.selected]="hero===selectedHero"
```

Angular Templates - Event Binding

https://angular.io/tutorial/toh-pt2#add-a-click-event-binding
https://angular.io/guide/template-syntax#event-binding

The parentheses, (), around ***click*** tell ***Angular*** to listen for the ***<li>*** element's click event. When the user clicks in the ***<li>***, ***Angular*** executes the onSelect(hero) expression in the components ***.ts*** file.



```
<li *ngFor="let hero of heroes" (click)="onSelect(hero)">
```

Angular dynamically changes the HTML ***template*** markup when the conditions change.

Modeling – Decorators

https://angular.io/guide/template-syntax#inputs-outputs
https://docs.angularjs.org/guide/decorators

Decorators are used to separate modification or decoration of a class without modifying the original source code. In AngularJS, *decorators* are functions that allow a service, directive or filter to be modified prior to its usage.

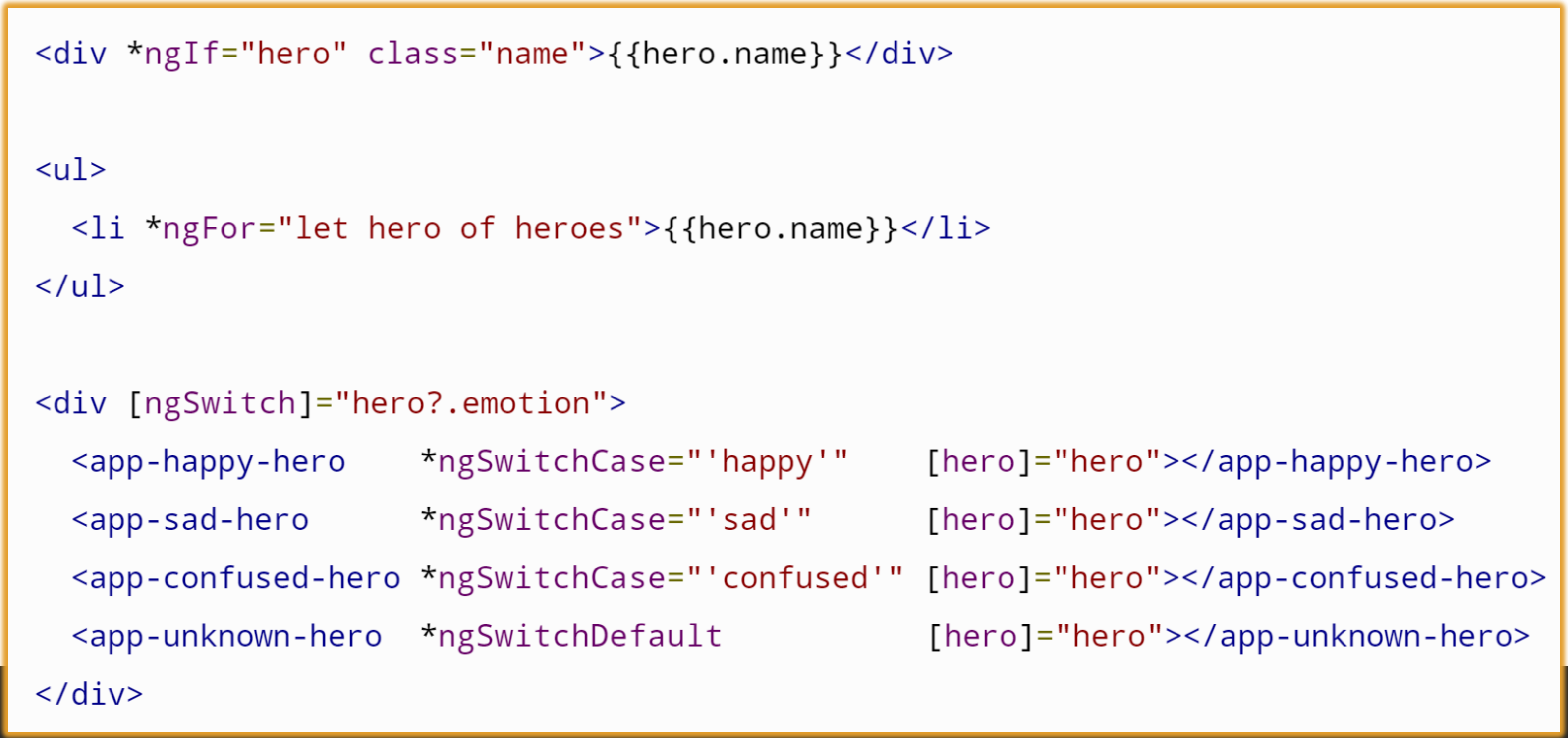
@Component - This indicates that the following class is a component. It provides metadata about the component, including its selector, templates, and styles.

- The selector identifies the component. The selector is the name you give the Angular component when it is rendered as an HTML element on the page. By convention, Angular component selectors begin with the prefix app-, followed by the component name.
- The template and style filenames reference the HTML and CSS files that StackBlitz generates.

Structural Directives

https://angular.io/api/common/NgIf
https://angular.io/api/common/NgForOf
https://angular.io/guide/template-syntax#ngSwitch
https://angular.io/guide/structural-directives

Structural directives shape or reshape the DOM's structure, typically by adding, removing, and manipulating the elements to which they are attached. Directives with an asterisk, *, are *structural directives*.



```
<div *ngIf="hero" class="name">{{hero.name}}</div>


<ul>
  <li *ngFor="let hero of heroes">{{hero.name}}</li>
</ul>


<div [ngSwitch]="hero?.emotion">
  <app-happy-hero    *ngSwitchCase="'happy'"    [hero]="hero"></app-happy-hero>
  <app-sad-hero      *ngSwitchCase="'sad'"      [hero]="hero"></app-sad-hero>
  <app-confused-hero *ngSwitchCase="'confused'" [hero]="hero"></app-confused-hero>
  <app-unknown-hero  *ngSwitchDefault           [hero]="hero"></app-unknown-hero>
</div>
```

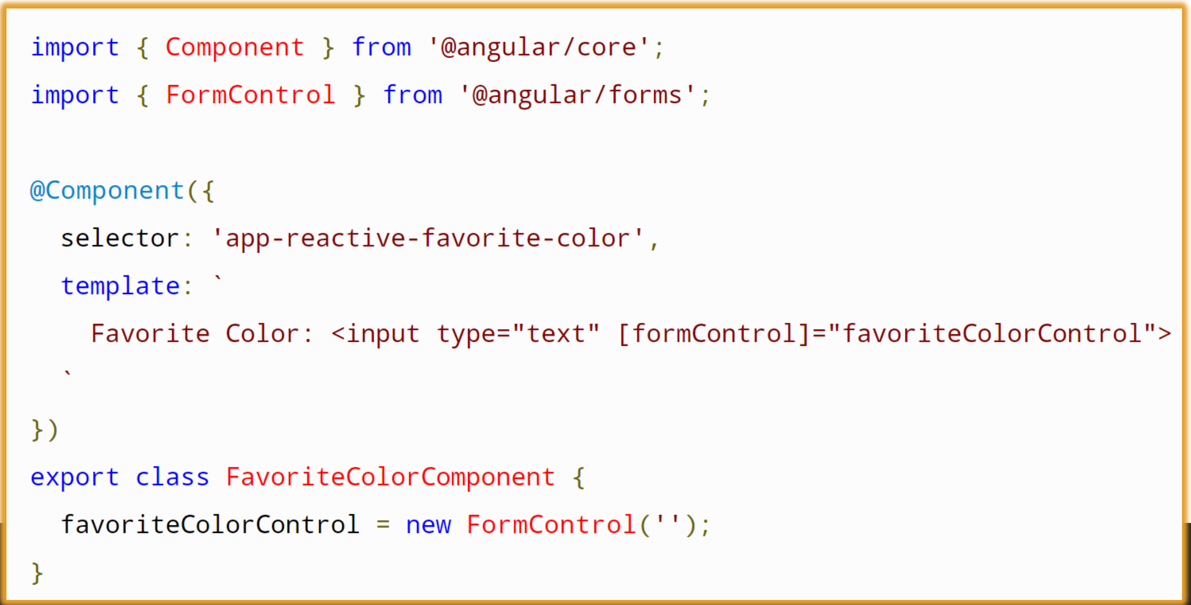
Angular Forms - Overview

https://angular.io/start/start-forms#forms-in-angular
https://angular.io/api/forms/FormBuilder
https://angular.io/guide/forms-overview

Angular provides two different approaches to handling user input through forms: ***reactive*** and ***template-driven***. Both capture user input ***events*** from the view (template), validate the user input, create a form model and data model to update, and provide a way to track changes.

- Reactive forms are more robust: they're more scalable, reusable, and testable. If forms are a key part of your application, use reactive forms.
- Template-driven forms are useful for adding a simple form to an app. They don't scale as well as reactive forms. If you have very basic form requirements and logic that can be managed solely in the template, use template-driven forms.

Reactive and ***template-driven forms*** both use a ***form model*** to track value changes between Angular forms and form input elements. The example to the right shows how the ***form model*** is defined and created.



```
import { Component } from '@angular/core';
import { FormControl } from '@angular/forms';

@Component({
  selector: 'app-reactive-favorite-color',
  template: `
    Favorite Color: <input type="text" [formControl]="favoriteColorControl">
  `
})
export class FavoriteColorComponent {
  favoriteColorControl = new FormControl('');
}
```

Reactive (Model-Driven) Forms

https://angular.io/start/start-forms#forms-in-angular
https://angular.io/api/forms/FormBuilder
https://angular.io/guide/forms-overview

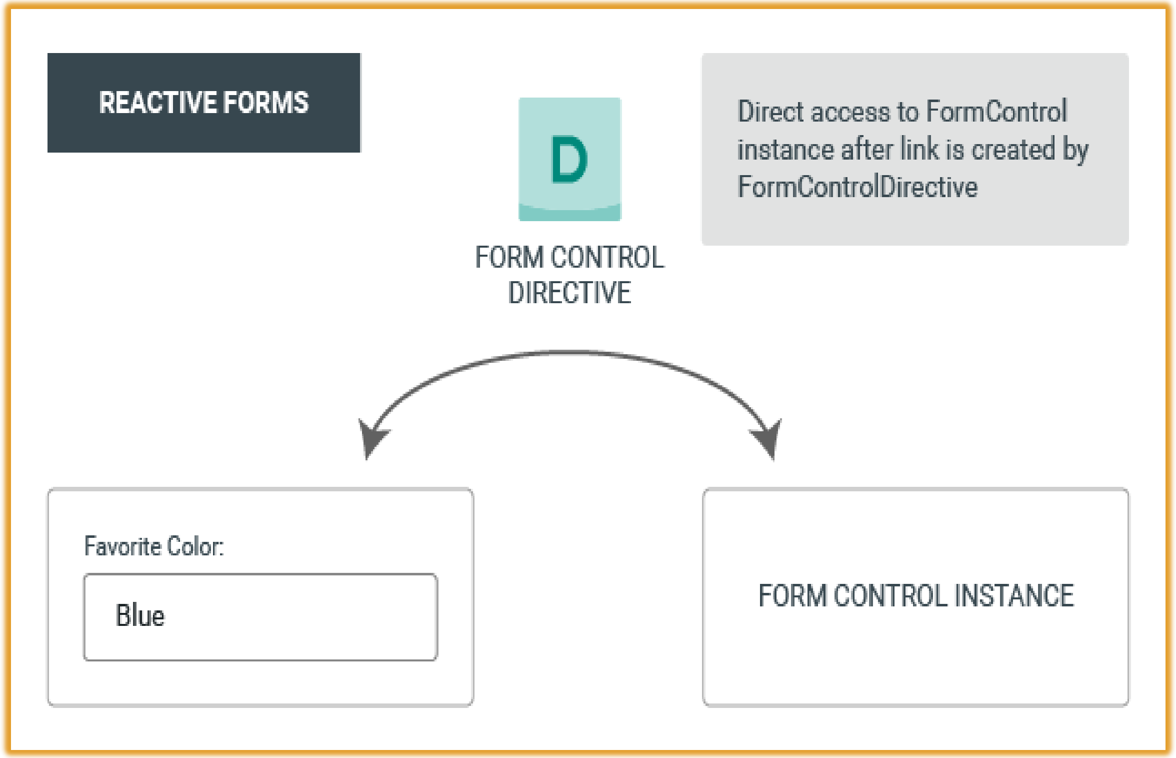
Reactive forms are built around observable streams, where form inputs and values are provided as streams of input values.

There are two parts to an ***Angular Reactive form***:

- the objects that live in the component to store and manage the ***form***, and
- the visualization of the ***form*** that lives in the ***template***.

The ***ReactiveFormsModule*** provides the ***FormBuilder*** service.

The ***form model*** is the "source of truth". The "source of truth" provides the value and status of the form element at a given point in time.


REACTIVE FORMS
D
FORM CONTROL DIRECTIVE
Direct access to FormControl instance after link is created by FormControlDirective
Favorite Color:
Blue
FORM CONTROL INSTANCE

Reactive Form Setup

https://angular.io/guide/reactive-forms#adding-a-basic-form-control

1. Import ReactiveFormsModule to ***app.module.ts*** with
   - import { ReactiveFormsModule } from '@angular/forms';.
2. Generate the new component with
   - ng generate component [ComponentName].
3. Import ***FormControl*** into the new component with
   - import { FormControl } from '@angular/forms';.
4. Set the initial value of the input field inside the component class declaration with
   - name = new FormControl('');.
5. Add the control to the view Template with
   - <input type="text" [formControl]="name">
6. You can add the component to any parent component view template with

- <app-name-editor></app-name-editor>

Now, whatever input you place in the input field will be transferred to the variable value in the component class.

```
import { ReactiveFormsModule } from '@angular/forms';
```

```
ng generate component NameEditor
```

```
import { Component } from '@angular/core';
import { FormControl } from '@angular/forms';
```

```
export class NameEditorComponent {
  name = new FormControl('');
}
```

```
<input type="text" [formControl]="name">
```

```
<app-name-editor></app-name-editor>
```

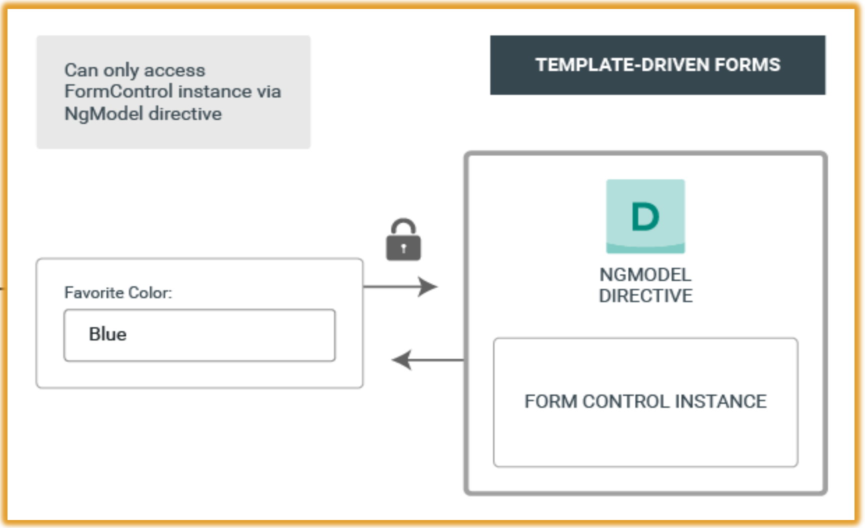
Template Driven Forms

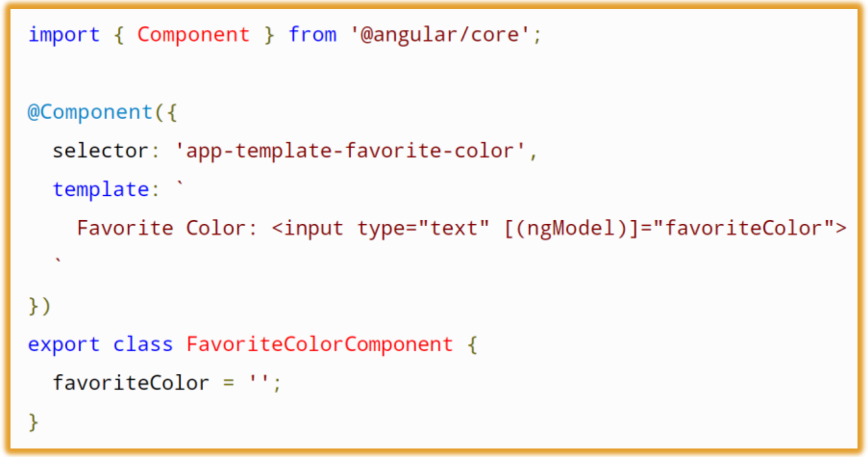
https://angular.io/start/start-forms#forms-in-angular
https://angular.io/api/forms/FormBuilder
https://angular.io/guide/forms-overview

You can build almost any form with an Angular template (login forms, contact forms).

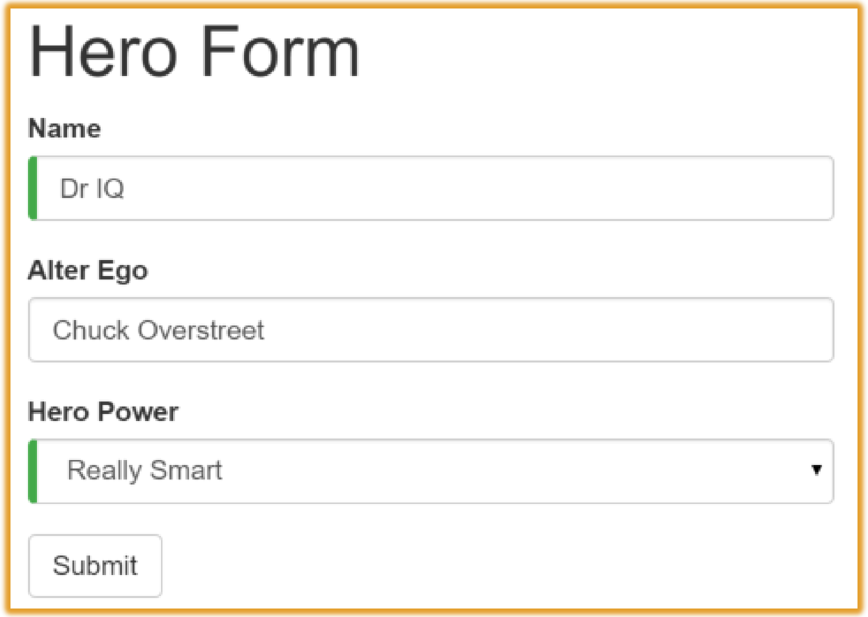
You can lay out the controls creatively, bind them to data, specify validation rules and display validation errors, and much more.

Angular makes the process easy by handling many of the repetitive, boilerplate tasks you'd otherwise wrestle with yourself.





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

@Component({
  selector: 'app-template-favorite-color',
  template: `
    Favorite Color: <input type="text" [(ngModel)]="favoriteColor">
  `
})
export class FavoriteColorComponent {
  favoriteColor = '';
}
```



Hero Form

Name

Dr IQ

Alter Ego

Chuck Overstreet

Hero Power

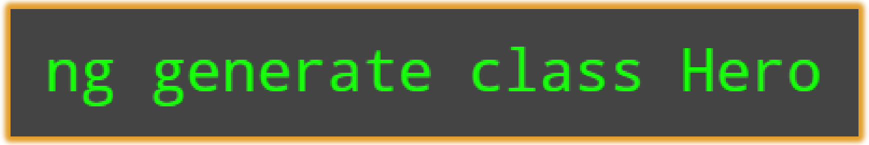
Really Smart

Submit

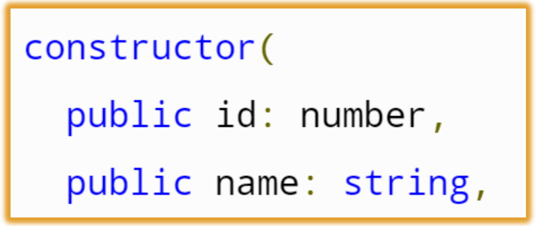
Template Driven Forms Setup

https://angular.io/guide/forms#introduction-to-template-driven-forms

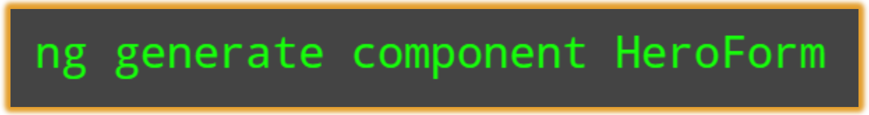
1. Create a new class with
   - ng generate class [className].
2. Add some properties like as id, etc.
3. Create a form Component with
   1. ng generate component HeroForm
4. Import the class into the new component with
   1. import { Hero } from '../hero';
5. Add an instance to the class model in the component with
   1. model = new Hero(18, 'Dr IQ’, etc……)



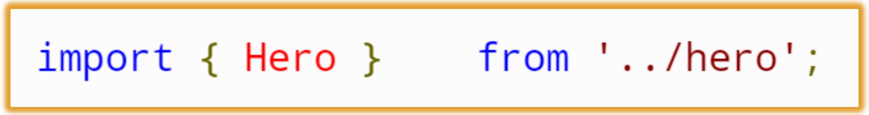
```
ng generate class Hero
```



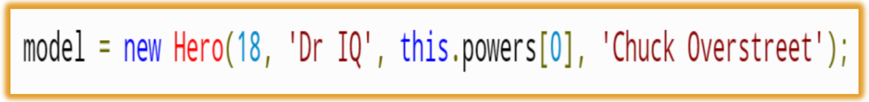
```
constructor(
  public id: number,
  public name: string,
```



```
ng generate component HeroForm
```



```
import { Hero }    from '../hero';
```

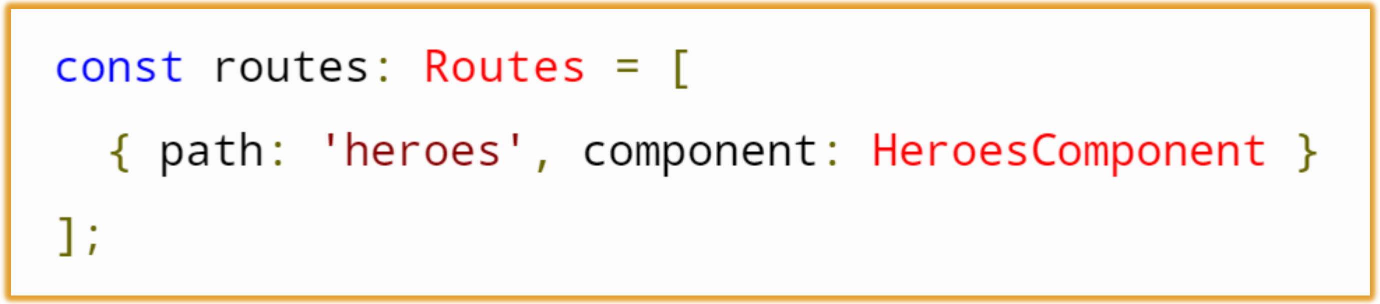


```
model = new Hero(18, 'Dr IQ', this.powers[0], 'Chuck Overstreet');
```

Angular Routing

https://angular.io/start/start-routing
https://angular.io/guide/router
https://angular.io/start/start-data#services

Register a route in ***app.module.ts***. A route associates one (or more) URL paths with a component.



```
const routes: Routes = [
  { path: 'heroes', component: HeroesComponent }
];
```

The ***RouterLink*** directive in the .html template gives the ***router*** control over the ***anchor*** element. Put ***RouterLink*** in the <a> element where you want to redirect to another (registered) URL.

Inject the ***ActivatedRoute*** into the constructor of the component where it will be used. It is specific to each routed component that the Angular Router loads. By injecting the ***ActivatedRoute***, you are configuring the component to use a service.

Angular Routing

https://angular.io/start/start-routing
https://angular.io/guide/router

Routes tell the ***Router*** which view to display when a user clicks a link.

A typical Angular ***Route*** has two properties:

- path: a string that matches the URL in the browser address bar.
- component: the component that the router should create when navigating to this route.

- The @NgModule (in ***app-routing.module.ts***) metadata initializes the router and starts it listening for browser location changes.

```
imports: [ RouterModule.forRoot(routes) ],
```

The forRoot() method supplies the service providers and directives needed for routing, and performs the initial navigation based on the current browser URL

Routing Step-by-step

https://angular.io/tutorial/toh-pt5#add-the-approutingmodule

1. Add a module called app-routing with
   - ng generate module app-routing --flat --module=app
2. Make sure RouterModule and Routes are imported into app-routing.module with
   - import { RouterModule, Routes } from '@angular/router';
   - Also import whatever ***component*** (from its relative location) you will be routing to into app-routing.module.ts
3. You can delete the CommonModule references and declarations array.
4. Configure routes in const routes: Routes = [{ path:'link', component: AssociatedComponent }]; in app-routing.module
5. Add imports: [RouterModule.forRoot(routes)], under @NgModule.
6. Also under @NgModule add exports: [RouterModule].
7. Add <a routerLink="/[link]">NameOfLink</a> to whatever page you want to add a link to.
8. Add …

Dependency Injection – Services and Injectables

https://angular.io/guide/glossary#dependency-injection-di
https://angular.io/guide/dependency-injection

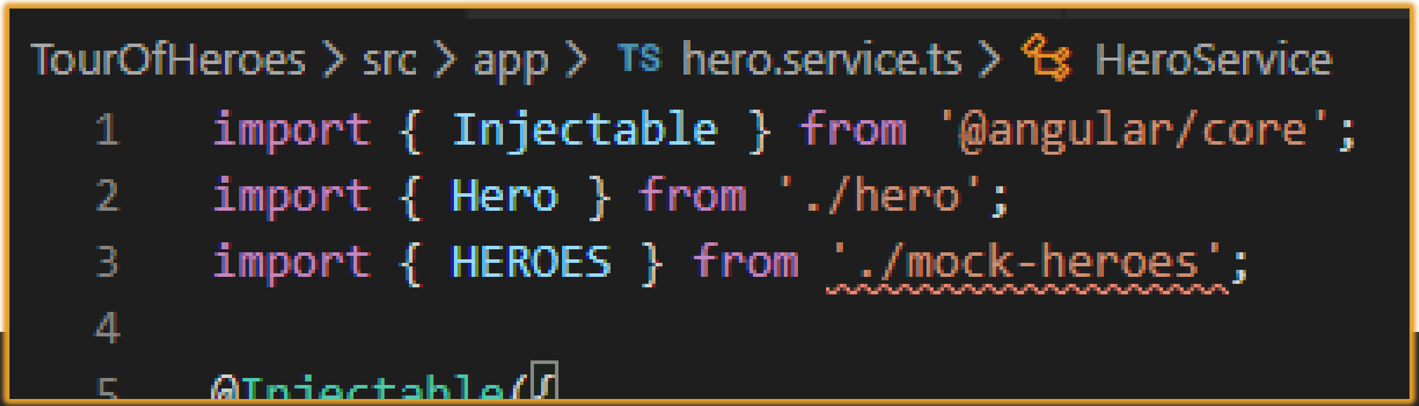
Components shouldn't fetch or save data directly. They should delegate data access to a ***Service***. A ***Service*** can get data from anywhere—a web service, local storage, or a mock data source.

Services are an integral part of Angular applications. In Angular, a ***service*** is an instance of a class that you can make available to any part of your application using Angular's ***dependency injection*** system.

Services are the place where you share data between parts of your application. The ***Service*** is your portal to persist data and have methods to access that data.

You can use ***services*** to share data across ***components.***

The @Injectable() decorator accepts a metadata object for the service, the same way the @Component() decorator does for component classes.



TourOfHeroes > src > app > TS hero.service.ts > HeroService

```
import { Injectable } from '@angular/core';
import { Hero } from './hero';
import { HEROES } from './mock-heroes';

@Injectable(
```

Dependency Injection – Services and Injectables

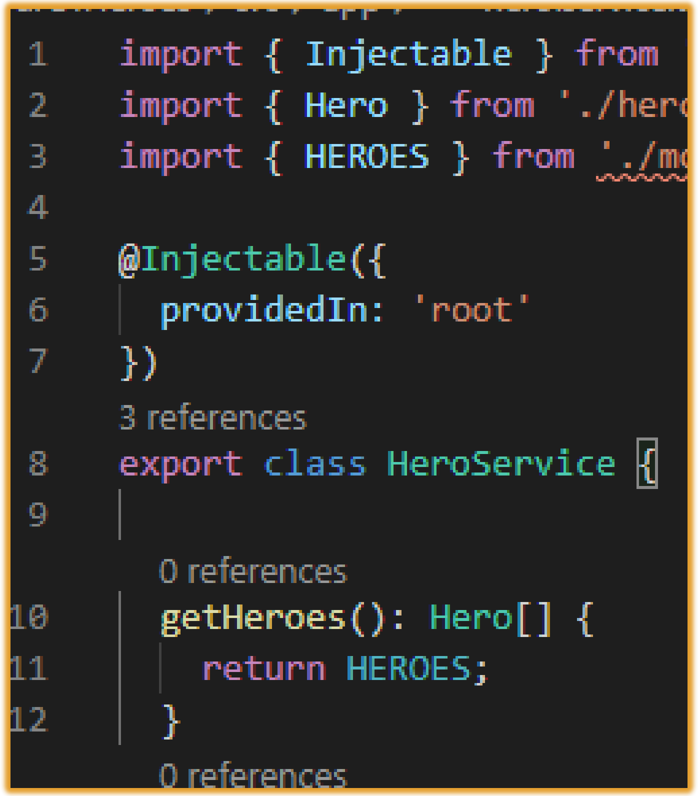
https://angular.io/tutorial/toh-pt4#provide-the-heroservice
https://angular.io/guide/dependency-injection

You must make the ***Service*** available to the ***dependency injection system*** before ***Angular*** can inject it into the ***Component*** by registering a ***provider***.

By default, ***the Angular CLI*** command ng generate service registers a ***provider*** with the ***root*** injector for your ***Service*** by including ***provider*** metadata that's provided in: 'root' in the @Injectable() ***decorator*** of the ***Service Component***.

When a ***Service*** is provided at the root level, Angular creates a single, shared instance of the ***Service*** and injects it into any class that asks for it.

Angular will also remove any unused ***Services***.



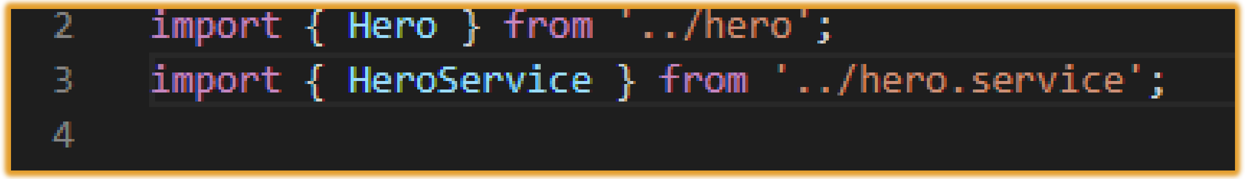
```
import { Injectable } from
import { Hero } from './her
import { HEROES } from './m

@Injectable({
  providedIn: 'root'
})
3 references
export class HeroService {

  0 references
  getHeroes(): Hero[] {
    return HEROES;
  }
  0 references
```

Angular – How to Use DI to Get a Service

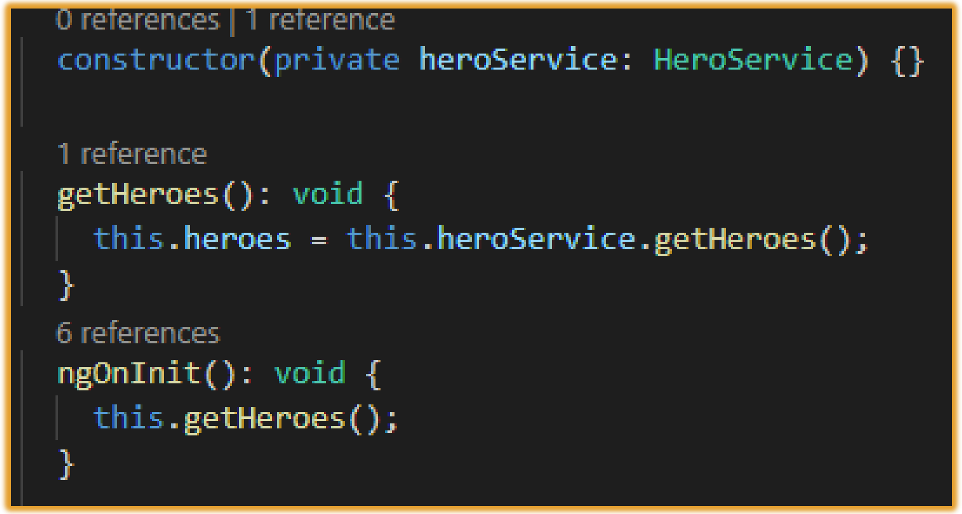
https://angular.io/tutorial/toh-pt4

To create a service to access your stored data,

1. Create a ***Service*** with
   - ng generate service [serviceName].
2. Import the ***Injectable*** symbol into the ***Service component***. This allows the ***Service*** to be injected into any other ***Component.***
   - import { Injectable } from '@angular/core';
3. Import the ***Service*** into the ***Component*** where it will be used with
   - import { ServiceName } from '../relative.location';.
4. Inject the ***Service*** into the constructor of the ***Component*** where it will be used with
   - constructor(private ServiceVariableName: ServiceName) {}.
5. Now you can access the ***Services*** functions with dot notation!



```
import { Hero } from '../hero';
import { HeroService } from '../hero.service';

```



```
0 references | 1 reference
constructor(private heroService: HeroService) {}

1 reference
getHeroes(): void {
  this.heroes = this.heroService.getHeroes();
}
6 references
ngOnInit(): void {
  this.getHeroes();
}
```

Best Practice is to use ngOnInit() to access and retrieve data from the service on instantiation of the Compoinent instead of retrieving it in the constructor.