## Notes

- The book assumes that you are working in a Unix-like environment. If you are on Windows you can use Cygwin so follow along with the bash terminal commands.

- All the project files for the book are hosted on github: https://github.com/st32lth/angular2-intro. You can clone the check out the project files. Throughout the book, you will see references to the project files. Those refer to this reposi example, `angular2-intro/project-files/hello-angular` refers to the `hello-angular` folder inside the `project-`

- Make sure you have `git` installed on your machine. That is, make sure you get an output for `git --version`.

- The book assumes that you have a working knowledge of JavaScript and Angular 1.x

- Node is heavily used throughout the book. Make sure that you follow the "Node" chapter to install Node and set pe correctly.

- All the keyboard shortcuts are mac-based. But if you are using a non-mac machine, you can almost always replace `ctrl` and you should be good. For example, if you a see a shortcut like `command + shift + b`, you can use `ctrl` where `ctrl` is obviously the `control` key.

## Installing Node

You can use nvm to install and manage Node on your machine. Copy the install script and run it:

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.30.1/install.sh | bash
```

After that, make a new terminal window and make sure that it is installed, by running:

```
nvm --help
```

Now you can use `nvm` to install Node `0.12.9` by running:

```
nvm install 0.12.9
```

After that, nvm is going to load version 0.12.9 automatically. If it doesn't, you can load it in the current shell, with:

```
nvm use 0.12.9
```

Note that you can load any node version in the current shell with `nvm use 0.x.y` after installing that version.

Also note that if you want to make `0.12.9` the default Node version on your machine, you can do so by running the

```
nvm alias default 0.12.9
```

Then you can verify that it is the default version by making a new terminal window and typing `node -v`.

## Permissions

Never use `sudo` to install packages, never do `sudo npm install <package>`. If you get permission errors while inst `sudo`, you can own the folders instead. So for example, if you get an error like:

```
Error: EACCES, mkdir '/usr/local'
```

you can own the folder with:

```
sudo chown -R `whoami` /usr/local
```

You can own folders until Node doesn't complain.

## Installing `live-server`

Install a package to verify that node is installed and everything is wired up correctly. We are going to use `live-serve`
book. So let's install that with:

```
npm i -g live-server
```

Then, you should be able to run `live-server` in any folder to serve the content of that folder:

```
mdkir ~/Desktop/sample && cd $_
live-server .
```

# Visual Studio Code

Visual Studio Code is a good IDE for developing web apps. In this chapter we will look at installing and configuring VS

## Visual Studio Code Basics

- Install Visual Studio Code from: https://code.visualstudio.com/
- You can open new projects by going to the `File > Open` tag, to etierh open a folder containing your project or a s
- Some useful keyboard shortcuts are:
    - `command + b` : to close/open the file navigator
    - `command + shift + p` : to open the prompt
- To install extensions open the prompt with `command + shift + p` and type:

```
> install extension
```

- You can change the keyboard shortcuts settings from `Preferences > Keyboard Shortcuts`. Open the settings an
add your own shortcuts:

```
// Place your key bindings in this file to overwrite the defaults
[
  {
    "key": "cmd+t",
    "command": "workbench.action.quickOpen"
  },
  {
    "key": "shift+cmd+r",
    "command": "editor.action.format",
    "when": "editorTextFocus"
  }
]
```

## Setting up VSCode for TypeScript

In this section we are going to set up Visual Studio Code for TypeScript. The project files for this chapter are in `angular2-intro/project-files/vscode`. You can either follow along or check out the folder to see the final result.

### Installing TypeScript

Before anything, we need to install the TypeScript compiler. You can install the TypeScript compiler with npm:

```
npm i typescript -g
```

Then to verify that it is installed, run `tsc -v` to see the version of the compiler. You will get an output like this:

```
message TS6029: Version 1.7.5
```

In addition to the compiler, we also need to install the TypeScript Definition manager for DefinitelyTyped (tsd). You car

```
npm i tsd -g
```

Using TSD, you can search and install TypeScript definition files directly from the community driven DefinitelyTyped re
verify that tsd is installed, run tsd with the `version` flag:

```
tsd --version
```

You should get an output like this:

```
>> tsd 0.6.5
```

After `tsd` and `tsc` are installed, we can compile a hello world program:

make a file called `hello.ts` on your desktop:

```
touch ~/Desktop/hello.ts
```

Then, put some TypeScript code in the file:

```
echo "const adder = (a: number, b: number): number => a + b;" > ~/Desktop/hello.ts
```

Then you can compile the file to JavaScript:

```
tsc ~/Desktop/hello.ts
```

It should output a file in `Desktop/hello.js`:

```
var adder = function (a, b) { return a + b; };
```

Now that your TypeScript compiler setup, we can move on to configuring Visual Studio Code.

## Add VSCode Configurations

- First download and install Visual Studio Code from the VSCode Website
- After installing VSCode, open it and then make a new window: `File > New Window`
- Then, make a folder on your desktop for a new project: `mkdir ~/Desktop/vscode-demo`
- After that, open the folder in VSCode: `File > open` and select the `vscode-demo` folder on your desktop.
- Now we need to make three configuration files:

  1. `tsconfig.json`: configuration for the TypeScript compiler
  2. `tasks.json`: Task configuration for VSCode to watch and compile files
  3. `launch.json`: Configuration for the debugger

The `tsconfig.json` file should be in the root of the project. Let's make the file and put the following in it:

```
{
  "compilerOptions": {
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true,
    "module": "commonjs",
    "target": "es5",
    "sourceMap": true,
    "outDir": "output",
    "watch": true
  }
}
```

Now to make the `tasks.json` file. Open the prompt with `command + shift + p` and type:

```
> configure task runner
```

Then put the following in the file and save the file:

```
{
  "version": "0.1.0",
  "command": "tsc",
  "showOutput": "silent",
  "isShellCommand": true,
  "problemMatcher": "$tsc"
}
```

The last thing that we need to set up is the debugger, i.e. the `launch.json` file. Right click on the `.vscode` folder ir navigator and make a new file called `launch.json` and put in the following:

```
{
  "version": "0.1.0",
  "configurations": [
    {
      "name": "TS Debugger",
      "type": "node",
      "program": "main.ts",
      "stopOnEntry": false,
      "sourceMaps": true,
      "outDir": "output"
    }
  ]
}
```

After you save the file, you should be able to see the debugger in the debugger dropdown options.

Now, we are ready to make the `main.ts` file in the root of the project:

**`main.ts`**

```
const sum = (a: number, b: number): number => a + b;
const r = sum(1,2);
console.log(r);
```

Now you can start the task to watch the files and compile as you work. Open the prompt with `command + shift + p`

```
> run build tasks
```

you can also use the `command + shift + b` keyboard shortcut instead. This will start the debugger and watch the fil a change to `main.ts`, you should be able to see the output in the `output` folder.

After the build task is running, we can put a breakpoint anywhere in our TypeScript code. Let's put a breakpoint on lir
on the margin. Then start the debugger by going to the debugger tab and clicking the green play icon.

Now you should see that the program will stop at the breakpoint and you should be able to step over or into your pro

To stop the task you can terminate it. Open the prompt and type:

```
> terminate running task
```

You can learn more about running TypeScript with VSCode on MSDN's blog.

## Running VSCode from the Terminal

If you want to run VSCode from the terminal, you can follow the guide on VSCode's website. Below is the summary c

**MAC**

Add the following to your "bash" file:

```
function code () { VSCODE_CWD="$PWD" open -n -b "com.microsoft.VSCode" --args $*; }
```

**Linux**

```
sudo ln -s /path/to/vscode/Code /usr/local/bin/code
```

**Windows**

You might need to log off after the installation for the change to the PATH environmental variable to take effect.

## Debugging App from VSCode

The "vscode-chrome-debug" extension allows you to attach VSCode to a running instance of chrome. This makes it v
because you can put breakpoints in your TypeScript code and run the debugger to debug your app. Let's get startec

In order to install the extension open the prompt in VSCode with `command + shift + p` and type:

```
> install extension
```

hit enter and then type:

```
debugger for chrome
```

Then just click on the result to install the extension. Restart VSCode when you are prompted.

After installing the extension, we need to update or create a `launch.json` file for debugging. You can create one in t
folder. After you created the file, put in the following:

```
{
  "version": "0.1.0",
  "configurations": [
    {
      "name": "Launch Chrome Debugger",
      "type": "chrome",
      "request": "launch",
      "url": "http://localhost:8080",
      "sourceMaps": true,
      "webRoot": ".",
      "runtimeExecutable": "/Applications/Google Chrome.app/Contents/MacOS/Google Chrome",
      "runtimeArgs": ["--remote-debugging-port=9222", "--incognito"]
    }
  ]
}
```

**Notes:**

- Depending on your platform you need to change the `runtimeExecutable` path to Chrome's executable path. After debugger you need to have a server running serving the app. You can change the `url` value accordingly. Also make `webRoot` path is set to the root of your web server.

- After that it is a good idea to close all the instances of chrome. Then, put a breakpoint in your code and run the deb everything is set up correctly, you should see an instance of chrome running in incognito mode. To trigger the breakp the page and you should be able to see the debugger paused at the breakpoint.

- Also make sure that you have the compiler running so that you can use the JavaScript output and the sourcemaps debugger. See the TypeScript and VSCode set up for more details.

# TypeScript Crash-course

In this chapter we will quickly go through the most important concepts in TypeScript so that you can have a better un Angular code that you will write. Knowing TypeScript definitely helps to understand Angular, but again it is not a requi project files for this chapter are in `angular2-intro/project-files/typescript` .

## TypeScript Basics

- TypeScript is a superset of JavaScript with additional features, among which optional types is the most notable. Th any valid JavaScript code (ES 2015/2016...) is valid TypeScript code. You can basically change the extension of the f compile it with the the TypeScript compiler.

- TypeScript defines 7 primary types:

  - boolean: `var isDone: boolean = false;`
  - number: `var height: number = 6;`
  - string: `var name: string = "bob";`
  - array: `var list:number[] = [1, 2, 3];` also `var list:Array<number> = [1, 2, 3];`
  - enum: `enum Color {Red, Green, Blue};`
  - any: `var notSure: any = 4;`
  - void: `function hello(): void { console.log('hello'); }`

## Interface

- An Interface is defined using the `interface` keyword
- Interfaces are used only during compilation time to check types
- By convention, interface definitions start with an `I` , e.g. : `IPoint`
- Interfaces are used in classical object oriented programming as a design tool
- Interfaces don't contain implementations
- They provide definitions only
- When an object implements an interface, it must adhere to the contract defined by the interface
- An interface defines what properties and methods an object must implement
- If an object implements an interface, it must adhere to the contract. If it doesn't the compiler will let us know.
- Interfaces also define custom types

### Basic Interface

Below is an example of an Interface that defines two properties and three methods that implementers should provide for:

```
interface IMyInterface {
  // some properties
  id: number;
  name: string;

  // some methods
  method(): void;
  methodWithReturnVal():number;
  sum(nums: number[]):number;
}
```

Using the interface above we can create an object that adheres to the interface:

```
let myObj: IMyInterface = {
  id: 2,
  name: 'some name',

  method() { console.log('hello'); },
  methodWithReturnVal () { return 2; },
  sum(numbers) {
    return numbers.reduce( (a,b) => { return a + b } );
  }
};
```

Notice that we had to provide values to **all** the properties defined by the Interface, and the implementations for **all** the
defined by the Interface.

And then of course you can use your object methods to perform operations:

```
let sum = myObj.sum([1,2,3,4,5]); // -> 15
```

## Classes as Interfaces

Because classes define types as well, they can also be used as interfaces. If you have an interface you can extend it
example:

```
class Point {
  x: number;
  y: number;
}
interface Point3d extends Point {
  z: number;
}
const point3d: Point3d = {x: 1, y: 2, z: 3};
console.log(point3d.x); // -> 1
```

First we are defining a class called `Point` that defines two fields. Then we define an interface called `Point3d` that e
`Point` by adding a third field. An then we create a point of type `point3d` and assign a value to it. We read the valu
`1`.

## Classes

- Classes are heavily used in classical object oriented programming
- It defines what an object is and what it can do

- A class is defined using the `class` keyword followed by a name
- By convention, the name of the class start with an uppercase letter
- A class can be used to create multiple objects (instances) of the same class
- An object is created from a class using the `new` keyword
- A class can have a `constructor` which is called when an object is made from the class
- Properties of a class are called instance variables and its functions are called the class methods
- Access modifiers can be used to make them public or private
- The instance variables are attached to the instance itself but not the prototype
- Methods however are attached to the prototype object as opposed to the instance itself
- Classes can inherit functionality from other classes, but you should favor composition over inheritance or make sure when to use it
- Classes can implement interfaces

Let's make a class definition for a car and incrementally add more things to it. The project files for this section are in `angular2-intro/project-files/typescript/classes/basic-class` .

## Adding an Instance Variable

The `Car` class definition can be very simple and can define only a single instance variable that all cars can have:

```
class Car {
  distance: number;
}
```

- `Car` is the name of the class, which also defines the custom type `Car`
- `distance` is a property that tracks the distance that car has traveled
- Distance is of type `number` and only accepts `number` type.

Now that we have the definition for a car, we can create a car from the definition:

```
let myCar:Car = new Car();
myCar.distance = 0;
```

- `myCar:Car` means that `myCar` is of type `Car`
- `new Car()` creates an instance from the `Car` definition.
- `myCar.distance = 0` sets the initial value of the `distance` to 0 for the newly created `car`

## Adding a Method

So far our car doesn't have any definitions for any actions. Let's define a `move` method that all the cars can have:

```
class Car {
  distance: number;
  move():void {
    this.distance += 1;
  }
}
```

- `move():void` means that `move` is a method that does not return any value, hence `void` .
- The body of the method is defined in `{ }`
- `this` refers to the instance, therefore `this.distance` points to the `distance` property defined on the car instar

- Now you can call the `move` method on the car instance to increment the `distance` value by 1:

```
myCar.move();
console.log(myCar.distance) // -> 1
```

## Using Access Modifiers

If you wanted to tell the compiler that the `distance` variable is private and can only be used by the object itself, you `private` modifier before the name of the property:

```
class Car {
  private distance: number;
  constructor () {
    ...
  }
  ...
}
```

- There are 3 main access modifiers in TypeScript: `private` , `public` , and `protected` :

- `private` modifier means that the property or the method is only defined inside the class only.
- `protected` modifier means that the property or the method is only accessible inside the class and the classes der class.
- `public` is the default modifier which means the property or the method is the accessible everywhere and can be a anyone.

## Adding a constructor

A `constructor` is a special method that gets called when an instance is created from a class. A class may contain a constructor declaration. If a class contains no constructor declaration, an automatic constructor is provided.

Let's add a constructor to the `Car` class that initializes the `distance` value to 0. This means that all the cars that ar this class, will have their `distance` set to 0 automatically:

```
class Car {
  distance: number;
  constructor () {
    this.distance = 0;
  }
  move():void {
    this.distance += 1;
  }
}
```

- `constructor()` is called automatically when a new car is created
- Parameters are passed to the constructor in the `()`
- The body of the constructor is defined in the `{ }`

Now, let's customize the car's constructor to accept `distance` as a parameter:

```
class Car {
  private distance: number;
  constructor (distance) {
    this.distance = distance;
  }
6 }
```

- On line 3 we are passing distance as a parameter. This means that when a new instance is created, a value should set the distance of the car.
- On line 4 we are assigning the value of distance to the value that is passed in

This pattern is so common that TypeScript has a shorthand for it:

```
class Car {
  constructor (private distance) {
  }
}
```

Note that the only thing that we had to do was to add `private distance` in the constructor parameter and remove `this.distance` and `distance: number`. TypeScript will automatically generate that. Below is the JavaScript output TypeScript:

```
var Car = (function () {
  function Car(distance) {
    this.distance = distance;
  }
  return Car;
})();
```

Now that our car expects a `distance` we have to always supply a value for the distance when creating a car. You ca values if you want so that the car is instantiated with a default value for the distance if none is given:

```
class Car {
  constructor (private distance = 0) {
  }
  getDistance():number { return this.distance; }
}
```

Now if I forget to pass a value for the `distance`, it is going to be set to zero by default:

```
const mycar = new Car();
console.log(mycar.getDistance()); //-> 0
```

Note that if you pass a value, it will override the default value:

```
const mycar = new Car(5);
console.log(mycar.getDistance()); //-> 5
```

## Setters and Getters (Accessors)

It is a very common pattern to have setters and getters for properties of a class. TypeScript provides a very simple sy that. Let's take our example above and add a setter and getter for the distance property. But before that we are goin `distance` to `_distance` to make it explicit that it is private. It is not required but it is a common pattern to prefix pri with an underscore.

```
class Car {
  constructor (private _distance = 0) {}
  getDistance():number { return this._distance; }
}
```

In order to create the getter method, we are going to use the `get` keyword and the name for the property followed b

```
class Car {
  constructor (private _distance = 0) {}
  get distance() { return this._distance; }
}
```

Now we can get the value of `distance` :

```
1 const car2 = new Car(5);
2 console.log(car2.distance) //-> 5
```

Note on line 2 that we didn't call a function. Behind the scenes, TypeScript creates a property for us, that's why it is r
Below is the relevant generated JavaScript:

```
Object.defineProperty(Car.prototype, "distance", {
  get: function () { return this._distance; },
  enumerable: true,
  configurable: true
});
```

JavaScript behind the scenes calls the get function for you to get the value, and that's why we simply did `car2.dist`
opposed to `car2.distance()` . For more information about `Object.defineProperty` checkout the MDN docs.

Similar to the getter, we can define a setter as well:

```
class Car {
  constructor (private _distance = 0) {}
  get distance() { return this._distance; }
  set distance(newDistance: number) { this._distance = newDistance; }
}
```

Now we can both get and set the distance value:

```
const coolCar = new Car();
console.log(coolCar.distance); // -> 0

coolCar.distance = 55;
console.log(coolCar.distance); // -> 55
```

Note that if we take out the setter, we won't be able to assign a new value to `distance` .

## Static Methods and Properties

Static methods and properties belong to the class but not the instances. For example, the `Array.isArray` method i
through the `Array` but not an instance of an array:

```
var x = [];
x.isArray // -> undefined
Array.isArray(x) //-> true
```

- On line 2 we are trying to access the `isArray` method, but obviously it is not defined because `isArray` is a stati
- On line three we are calling the static `isArray` method from `Array` and we can check if `x` is an array.

If you look at the Array documentation you can see that methods and properties are either defined on the `Array.pro` `Array`:

- `Array.prototype.x`: makes `x` available to all the instances of `Array`
- `Array.x`: `x` is static and only available through `Array`.

Now that we have some context, let's see how you can define static methods and properties in TypeScript. Consider

```
class Car {
  static controls: {isAuto: boolean } = {
    isAuto: true
  };
  static isAuto():boolean {
    return Car.controls.isAuto;
  }
  constructor (private _distance = 0) {}
  get distance() { return this._distance; }
}

console.log(Car.controls); // -> { isAuto: true }
console.log(Car.isAuto()); // -> true
```

- On line 2 we are defining a static property called `controls` using the `static` modifier. Then we specify the form a value for it.
- On line 5 we are defining a static method called `isAuto` using the the `static` modifier. This method simply retur `isAuto` from the static `control` object. Not that we get access to the class using the name of the class as oppose `this`. i.e. `return Car.controls.isAuto`

## Implementing an Interface

Classes can implement one or multiple interfaces. We can make the `Car` class implement two interfaces:

```
interface ICarProps {
  distance: number;
}
interface ICarMethods {
  move():void;
}
```

Making the `Car` class implement the interfaces:

```
class Car implements ICarProps, ICarMethods {
  distance: number;
  constructor () {
    this.distance = 5;
  };
  move():void {
    this.distance += 1;
  };
}
```

The above example is silly, but it shows the point that a class can implement one or more interfaces. Now if the class provide implementations for any of the interfaces, the compiler will complain. For example, if we leave out the `distar` variable, the compiler will print out the following error:

> error TS2420: Class 'Car' incorrectly implements interface 'ICarProps'. Property 'distance' is missing in type 'Car'.

## Inheritance

In Object-oriented programming, a class can inherit from another class which helps to define shared attributes and m objects. Although this pattern is very useful, it should be used cautiously as it can lead to code that is hard to maintai more about classical inheritance and prototypical inheritance by watching Eric Elliot's talk at O'Reilly's Fluent Confere files for this section are in `angular2-intro/project-files/typescript/classes/inheritance` .

Let's get started by creating a base class called `Vehicle` . This class is going to be the base class for other classes later.

```
// Vehicle.ts
export class Vehicle {
  constructor( private _name: string = 'Vehicle',
               private _distance: number = 0 ) { }
  get distance(): number { return this._distance; }
  set distance(newDistance: number) { this._distance = newDistance; }
  get name(): string { return this._name;}
  set name(newName: string) { this._name = newName; }
  move() { this.distance += 1 }
  toString() { return this._name; }
}
```

There is nothing special in this class. We are just creating a class that has two private properties (name, distance) anc the setters and getters for them. Additionally, we are defining the `toString` method that JavaScript internally calls ir contexts". The constructor is the most notable of all the other methods:

- It sets the `name` property to "Vehicle" for all the instances
- It also sets the `distance` property to 0.

This means that when a class extends the `Vehicle` class, it will have to call the constructor of `Vehicle` using the Let's do that now by creating two classes called `Car` and `Truck` that inherit from the `Vehicle` class:

**`cars.ts`**

```
import {Vehicle} from './vehicle';
export class Car extends Vehicle {
  constructor(name?: string) {
    super();
    this.name = name || 'Car';
  }
}
export class Truck extends Vehicle {
  constructor(name?: string) {
    super();
    this.name = name || 'Truck';
  }
}
```

- The `Car` class and the `Truck` class both look almost identical. They both inherit from the `Vehicle` using the `ex` keyword.
- They both call the `Vehicle` 's constructor in their own constructor method before implementing their own: `constructor(name?: string) { super(); }`
- They both take an optional `name` property to set the name of the vehicle. If not name is provided, it will be set to e 'Truck'

Now let's create the `main` file and run the file:

```
1  import {Car, Truck} from './cars';
2
3  /**
4   * Creating a new car from `Car`
5   */
6  const car = new Car();
7  console.log(car.name);
8  car.distance = 5;
9  car.move();
10 car.move();
11 console.log(car.distance);
12 /**
13  * Creating a new Truck.
14  */
15 const truck = new Truck();
16 console.log(truck.name);
```

- On line 1 we are importing the `Car` and the `Truck` class.
- and then we create a `Car` and `Truck` instance and log their names and distance to the console.

Run the build task (command + shift + b) and run the file (F5) and you should see the output:

```
node --debug-brk=7394 --nolazy output/main.js
Debugger listening on port 7394
Car
7
Truck
```

You can play around with the code above an try passing a string when instantiating a `Car` or a `Truck` to see the na

**TODO**

- `constructor overloading`

## Class Decorators

There are different types of decorators in TypeScript. In this section we are going to focus on Class Decorators.

**TODO**

`add content`

# Modules

- In TypeScript you can use modules to organize your code, avoid polluting the global space, and expose functionalit use.
- Multiple modules can be defined in the same file. However, it makes more sense to keep on module per file
- If you want, you can split a single module across multiple files
- If you decide to split a module across different files, this is how you would do it:
  - Create the module file: `mymodule.ts` and declare your module there: `module MyModule {}`
  - Create another file: `mymodule.ext1.ts` and on top of the file add: `/// <reference path="mymodule.ts" />`. Then in the the same name of the module and add more stuff to it: `module MyModule { // other stuff... }`
  - Then in your main file, you need two things on top of the file:
    - `/// <reference path="mymodule.ts" />`
    - `/// <reference path="mymodule.ext1.ts" />`
  - Then, you can use the name of your module to refer to the symbols defined: `MyModule.something`, `MyModule.somethingE`

- TypeScript has two system: one used internally and the other used externally
- External modules are used if your app uses CommonJS or AMD modules. Otherwise, you can use TypeScript's inte system
- Using TypeScript's internal module system, you can:
    - use the `module` keyword to define a module: `module MyModule { ... }`
    - split modules into different files that contribute to a single module
    - use the `/// <reference path="File.ts" />` tag to tell the compiler how files are related to each other when modules are
- Using TypeScript's external module system:
    - you cannot use the `module` keyword. The `module` keyword is used only by the internal module system.
    - instead of the `reference` tag, you can use the `import` keyword to define the relationship between modules
    - you can import symbols using the file name: `import mymodule = require('mymodule')`

The project files for this chapter are in `angular2-intro/project-files/typescript/modules`.

## Simple Module

Let's create a simple module that contains two classes. The first class is a vehicle class and the second is a car class from the vehicle class. Then we are going to expose the car class to the outside world and import it from another file. for this section are in `angular2-intro/project-files/typescript/modules/basic-module`.

First, create the `main.ts` file and copy paste the following:

**`main.ts`**

```
1  module MyModule {
2    class Vehicle {
3      constructor (public name: string = 'Vehicle', private _distance: number = 0) {}
4      get distance():number { return this._distance; }
5      set distance(newDistance: number) { this._distance = newDistance; }
6      move() { this.distance += 1 }
7    }
8  }
```

- On line 1 we are defining the module called `MyModule`.
- Inside this module we have defined a class called `Vehicle` that has a distance property and a setter and getter.

Now we want to create a class and export it so that it can be imported by others:

**`main.ts`**

```
1   module MyModule {
2     class Vehicle {
3       constructor (public name: string = 'Vehicle', private _distance: number = 0) {}
4       get distance():number { return this._distance; }
5       set distance(newDistance: number) { this._distance = newDistance; }
6       move() { this.distance += 1 }
7     }
8     // -> adding the car class
9     export class Car extends Vehicle {
10      constructor (public name: string = 'Car') {
11        super();
12      }
13    }
14  }
```

- On line 9 we are using the `export` keyword to indicate that the `Car` class is exposed and can be used by others

Now, let's create a car using the `Car` class defined in the `MyModule` module:

```
const mycar = new MyModule.Car('My Car');
console.log(mycar.name);
```

Note that we accessed the `Car` class using the `MyModule` symbol: `MyModule.Car`. Now we can split up the modu
file and import it into the main file. Let's create a file called `MyModule.ts` and move the module definition to that file.
file we are just going to import the module and use the car class from it.

**`main.ts`**

```
/// <reference path="MyModule.ts" />
const mycar = new MyModule.Car('My Car');
console.log(mycar.name);
```

Note that we can create an alias to the `MyModule` using `import AliasName = MyModule`. Now you can reference th
with `AliasName`:

```
/// <reference path="MyModule.ts" />
import AliasName = MyModule;
const mycar = new AliasName.Car('My Car');
console.log(mycar.name);
```

Now if we run this in debug mode, the compiler will complain that it can't find the `MyModule` reference. Because of tl
make some changes to our config files. First, we are going to add the `out` property in the `tsconfig.json` file. This
compiler to compile all the files into a single file:

```
"out": "output/run.js",
```

So our `tsconfig.json` file will look like this:

```
{
  "compilerOptions": {
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true,
    "module": "commonjs",
    "target": "es5",
    "sourceMap": true,
    "outDir": "output",
    "out": "output/run.js",
    "watch": true
  }
}
```

Now if you run the build, you should see that all the project has been compiled into `output/run.js`. In addition to th
`tsconfig.json` file, we are going to update the `launch.json` file and add a new configuration field:

```
{
  "name": "TS All Debugger",
  "type": "node",
  "program": "output/run.js",
  "stopOnEntry": false,
  "sourceMaps": true
}
```

Now we should be able to use the debugger and put breakpoints in our TypeScript files. Select `TS All Debugger` fr
debugger dropdown and run the debugger and it should stop if you put a breakpoint in any of your TypeScript files.

**NOTE** Using the configuration files above we can compile all the TypeScript files into a single JavaScript file. But som
not what you want. Be aware that using the above configuration you will not get an output for each TypeScript file.

## Splitting Internal Modules

Internal modules in TypeScript are open ended. This means that you can define a module with the same name in diffe keep adding to it. This is also known as merging. In this section we are going to demonstrate merging multiple files th a single module called `Merged` . The project files for this section are in `angular2-intro/project-files/typescript/modules/merged-module` .

First, we are going to make two files: `A.ts` and `B.ts` . In each file we are going to define the `Merged` module:

```
// A.ts
module Merged {
  const name = 'File A'; // not exported
  export class Door {
    constructor (private _color = 'white') {}
    get color() { return this._color; }
    set color(newColor) { this._color = newColor; }
  }
}
```

and then the `B.ts` file:

```
// B.ts
module Merged {
  const name = 'File B'; // not exported
  export class Car {
    constructor(public distance = 0) {}
    move () {this.distance += 1;}
  }
}
```

We just created two files called `A.ts` and `B.ts` and each file we defined the `Merged` module and added a class to exported it. Now we are going to make the `main.ts` file and reference these two files:

```
// main.ts
/// <reference path="./A.ts" />
/// <reference path="./B.ts" />
```

And now we can use the classes defined in the `Merged` module, that is the `Car` and the `Door` class:

```
/// <reference path="./A.ts" />
/// <reference path="./B.ts" />
const car: Merged.Car = new Merged.Car();
const door: Merged.Door = new Merged.Door();
door.color = 'blue';
car.move();
car.move();
console.log(car.distance);
console.log(door.color);
```

if you run the build task (command + shift + b) and hit F5 you should see the following output:

```
node --debug-brk=19237 --nolazy output/run.js
Debugger listening on port 19237
2
blue
```

## External Modules

In addition to TypeScript's internal module system, you can use external modules as well. In this section we are going
how you can use external modules in TypeScript. The project files for this section are in
`angular2-intro/project-files/typescript/modules/external-module` .

Let's say I have a JavaScript Node module defined in CommonJS format in a file called `common.js` :

```
// common.js
module.exports = function () {
  this.name = 'CommonJS Module';
};
```

In order to import this we need to do two things: first, we need to install Node's Type Definitions. Then we need to re
module. To install Node's Type Definitions run the following the terminal in the root of your project:

```
tsd install node --save
```

Now you should see a folder called `typings` containing the type definitions. Now that we have Node's type definitio
reference to it on top of `main.ts` :

```
// main.ts
/// <reference path="./typings/node/node.d.ts" />
```

and then we are going to require the module and log it to the console:

```
// main.ts
/// <reference path="./typings/node/node.d.ts" />
const common = require('./common');
console.log(common()); // --> CommonJS Module
```

After running the build task ( command + shift + b ), and running the file (F+5) you should see the following output:

```
node --debug-brk=32221 --nolazy run.js
Debugger listening on port 32221
CommonJS Modules
```

**Note** the configuration files that we are using:

**`tsconfig.json`**

```
{
  "compilerOptions": {
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true,
    "module": "commonjs",
    "target": "es5",
    "sourceMap": true,
    "outDir": "output",
    "out": "run.js",
    "watch": true
  }
}
```

**`launch.json`**

```
{
  "version": "0.1.0",
  "configurations": [
    {
      "name": "TS All Debugger",
      "type": "node",
      "program": "./run.js",
      "stopOnEntry": false,
      "sourceMaps": true
    }
  ]
}
```

## Decorators

- Decorators can be used to add additional properties and methods to existing objects.
- Decorators are a declarative way to add metadata to code.
- There are four decorators: ClassDecorator, PropertyDecorator, MethodDecorator, ParameterDecorator
- TypeScript supports decorators and does not know about Angular's specific annotations.
- Angular provides annotations that are made with decorators behind the scenes

## Method Decorators

Goals: - make a method decorator called `log`. - Decorate `someMethod` in a class using `@log`

```
class SomeClass {
  @log
  someMethod(n: number) {
    return n * 2;
  }
}
```

In the usage, `someMethod` has been decorated with `log` using the `@` symbol. `@log` is decorating `someMethod` b
placed right before the method.

- Decorator Implementation:

```
function log(target: Function, key: string, value: any) {
  return {
    value: function (...args: any[]) {
      var a = args.map(a => JSON.stringify(a)).join();
      var result = value.value.apply(this, args);
      var r = JSON.stringify(result);
      console.log(`Call: ${key}(${a}) => ${r}`);
      return result;
    }
  };
}
```

A method decorators takes a 3 arguments:

- `target`: the method being decorated.
- `key`: the name of the method being decorated.
- `value`: a property descriptor of the given property if it exists on the object, undefined otherwise. The property des
obtained by invoking the `Object.getOwnPropertyDescriptor` function.

**TODO**

- Add decorator content for each type.

# Angular

This chapter will walk you through the main concepts in Angular. We will start by looking at components, and then we
pipes, services, events and other concepts. By the end of the chapter you should have a basic understanding of the r
concepts in Angular.

The goal of this chapter is to get your feet wet without scaring you with a lot of details. Don't worry, there will be a lot
chapters.

## Project Files

### Running the Project Files

First, make sure that you have cloned the code repo somewhere on your machine:

```
cd ~/Desktop && git clone git@github.com:st321th/angular2-intro.git
```

In order to run the project files, you need to do two things:

- First, install the server dependencies and run the server in the root of the repo:

```
cd angular2-intro && npm i && npm start
```

After the dependencies are installed, it will open up the browser at port 8080.

- The next step is to install the dependencies for angular examples. Go to `project-files/angular-examples` and ir
dependencies:

```
cd project-files/angular-examples && npm i
```

After following the steps above, you should be able to see the examples in the browser. For example, if you want to s
`basic-component` example, you can go to the following url:

```
http://localhost:8080/project-files/angular-examples/basic-component/index.html
```

### Starter Project

There is a starter project in `angular-examples/starter`. You can make a copy of that folder if you want to work on s
The steps for running the project is the same for all the projects:

- Install the dependencies for the dev server in the root of the repo with `npm i` **(needed once)**
- Start the dev server in the root of the repo with `npm start`
- Install the dependencies for angular examples: `cd project-files/angluar-examples && npm i` **(needed once)**
- Open your project in VSCode: `code project-files/angular-examples/starter`
  - Close all chrome instances (quit out of Chrome)
  - In VSCode start the build with `command + shift + b` and run the app by hitting F5 on the keyboard
- If you don't want to use VSCode, you can use any other editor that you want. But make sure that you run the Type
in the project folder: `cd project-files/angular-examples/starter && tsc -w`.

## Using the Docs

Angular API reference can be found at: https://angular.io/docs/ts/latest/api.

If you are looking for annotations or decorators, look for the keyword followed by `metdata`. For example, if you want

Component decorator, you would look for: `ComponentMetadata` . Below are the common metadata class names:

- `ComponentMetadata`
- `DirectiveMetadata`
- `PipeMetadata`
- `InjectMetadata`
- `InjectableMetadata`

**TODO**

**Common Interfaces**

- `OnInit`

**TODO**

**Common Enums**

- `ChangeDetectionStrategy`

**TODO**

# Metadata Classes Cheatsheet

- Angular uses Metadata to decorate classes, methods and properties.
- The most notable Metadata is the `@component` Metadata.
- Metadta classes are very convenient and they make it easy to work with components, services and the dependenc system

Below is a list of Angular's core Metadata classes categorized under directives/components, pipes and di.

**Directive/component Meta-data**

- Component: used to define a component
  - View: used to define the template for a component
  - ViewChild: used to configure a view query
  - ViewChildren: used to configure a view query
- Directive: used to define a directive
  - Attribute used to grab the value of an attribute on an element hosting a directive
  - ContentChild: used to configure a content query
  - ContentChildren: used to configure a content query
  - Input: used to define the input to a directive/component
  - Output: used to define the output events of a directive/component
  - HostBinding: used to declare a host property binding
  - HostListener: used to declare a host listener

**Pipes**

- Pipe: used to declare reusable pipe function

**DI**

- Inject: parameter metadata that specifies a dependency.
- Injectable: a marker metadata that marks a class as available to Injector for creation.

- Host: Specifies that an injector should retrieve a dependency from any injector until reaching the closest host.
- Optional: parameter metadata that marks a dependency as optional
- Self: Specifies that an Injector should retrieve a dependency only from itself.
- SkipSelf: Specifies that the dependency resolution should start from the parent injector.
- Query: Declares an injectable parameter to be a live list of directives or variable bindings from the content children c
- ViewQuery: Similar to `QueryMetadata`, but querying the component view, instead of the content children.

**TODO**

# Component Basics

- Technically speaking components are directives that extend directives with views
- A component encapsulates a specific piece of functionality and components work together to deliver app's functior
- Generally speaking, every app has a root component that bootstraps the application. And when the app is bootstra
starts from the root component and resolves the sub trees of components

In this section we are going to write a simple `HelloAngular` component, compile it and run it in the browser. In addi
configure VSCode to build the TypeScript files as we go.

Note that there is a lot to talk about components. We are going dive into components a lot more in later chapters, bu
just keep things simple.

The project files for this chapter are in `angular2-intro/project-files/angular-examples/basic-component` You ca
along or just look at the final result

In order to run the project files, please refer to the Running the Project Files section.

### Getting Started

Make a folder on your desktop called `hello-angular` and navigate to it:

```
mkdir ~/Desktop/hello-angular && cd $_
```

Start npm in this folder with `npm init` and accept all the defaults.

After that, add the `dependencies` and `devDependencies` field to your `package.json` file:

```
"dependencies": {
  "angular2": "^2.0.0-beta.1",
  "es6-promise": "^3.0.2",
  "es6-shim": "^0.33.3",
  "reflect-metadata": "0.1.2",
  "rxjs": "5.0.0-beta.0",
  "zone.js": "0.5.10"
},
"devDependencies": {
  "systemjs": "^0.19.16"
}
```

your `package.json` file should look something like the follwoing:

```
{
  "name": "hello-angular",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Stealth <st321th@gmail.com> (http://github.com/st321th)",
  "license": "ISC",
  "dependencies": {
    "angular2": "^2.0.0-beta.1",
    "es6-promise": "^3.0.2",
    "es6-shim": "^0.33.3",
    "reflect-metadata": "0.1.2",
    "rxjs": "5.0.0-beta.0",
    "zone.js": "0.5.10"
  },
  "devDependencies": {
    "systemjs": "^0.19.16"
  }
}
```

Then run `npm i` to install the dependencies.

After all the dependencies are installed, start VSCode in this folder with `code .`

Then create a `index.html` file in the root of the project and put in the following:

**`index.html`**

```html
<html>
<head>
  <title>Hello Angular</title>

  <script src="/node_modules/angular2/bundles/angular2-polyfills.js"></script>
  <script src="/node_modules/systemjs/dist/system.src.js"></script>
  <script src="/node_modules/rxjs/bundles/Rx.js"></script>
  <script src="/node_modules/angular2/bundles/angular2.dev.js"></script>

  <!-- add systemjs settings later -->

</head>

<body>
  <!-- add app stuff later -->
</body>

</html>
```

This loads all the necessary scripts that we need to run Angular in the browser.

### Note

If you need to support older browsers, you need to include the `es6-shims` before everything else:

```html
<script src="/node_modules/es6-shim/es6-shim.js"></script>
```

## Making a Simple Component

Let's start by making the `main.ts` file in the root of the project. In this file we are going to define the main componer `HelloAngular` and then bootstrap the app with it:

**`main.ts`**

```
1  import {Component, OnInit} from 'angular2/core';
2  import {bootstrap} from 'angular2/platform/browser';
3
4  @Component({
5    selector: 'app',
6    styles: [`h1 { line-height: 100vh; text-align: center }`],
7    template: `<h1>{{ name }}</h1>`
8  })
9  class HelloAngular implements OnInit {
10   name: string;
11   constructor() { this.name = 'Hello Angular'; }
12   ngOnInit() { console.log('component linked'); }
13 }
14
15 bootstrap(HelloAngular, []);
```

- On line 1 we are importing the `component` meta data (annotation) and the `onInit` interface.
- On line 2 we are loading the `bootstrap` method that bootstraps the app given a component.
- On line 4, we are defining a component using the `component` decorator. The `@component` is technically a class de because it precedes the `HelloAngular` class definition.
- On line 5, we are telling angular to look out for the `app` tag. So when Angular looks at the html and comes across `<app></app>` tag, it is going to load the template (on line 6) and instantiates the class for it (defined on line 9).
- On line 9, we are defining a class called `HelloAngular` that defines the logic of the component. And for fun, we ar the `OnInit` interface to log something to the console when the component is ready with its data. We will learn more lifeCycle hooks later.
- Last but not least, we call the `bootstrap` method with the `HelloAngular` class as the first argument to bootstrap the `HelloAngular` component.

**Compiling the Component**

Now we need to compile the file to JavaScript. We can do it from the terminal, but let's stick to VSCode. In order to t make two config files:

1. First is the standard `tsconfig.json` file

2. And the `tasks.json` file for VSCode to do the compiling

Create the `tsconfig.json` file in the root of the project and put in the following:

**`tsconfig.json`**

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "system",
    "moduleResolution": "node",
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "removeComments": false,
    "noImplicitAny": false,
    "outDir": "output",
    "watch": true
  },
  "exclude": [
    "node_modules"
  ]
}
```

Then create the `tasks.json` in the `.vscode` folder in the root of the project and put in the following:

**`.vscode/tasks.json`**

```
{
  "version": "0.1.0",
  "command": "tsc",
  "showOutput": "silent",
  "isShellCommand": true,
  "problemMatcher": "$tsc"
}
```

- Now we can build the TypeScript files as we work. We just need to start the build task with `command + shift + b` prompt. If you want to use the prompt do the following:
  - Use `command + shift + p` to open the prompt
  - Then, type `> run build task` and hit enter to start the build task.
- After you run the build task, you should see an `output` file generated with `main.js` and the source maps in it.
- The task is watching the files and compiling as you go. To stop the task, open the prompt and type:

```
> terminate running task
```

### Loading the Component

After compiling the component, we need to load it to the `index.html` file with `Systemjs`. Open the `index.html` fil `<!-- add systemjs settings later -->` with the following:

```
<script>
  System.config({
    packages: {
      output: {
        format: 'register',
        defaultExtension: 'js'
      }
    }
  });
  System.import('output/main')
  .then(null, console.error.bind(console));
</script>
```

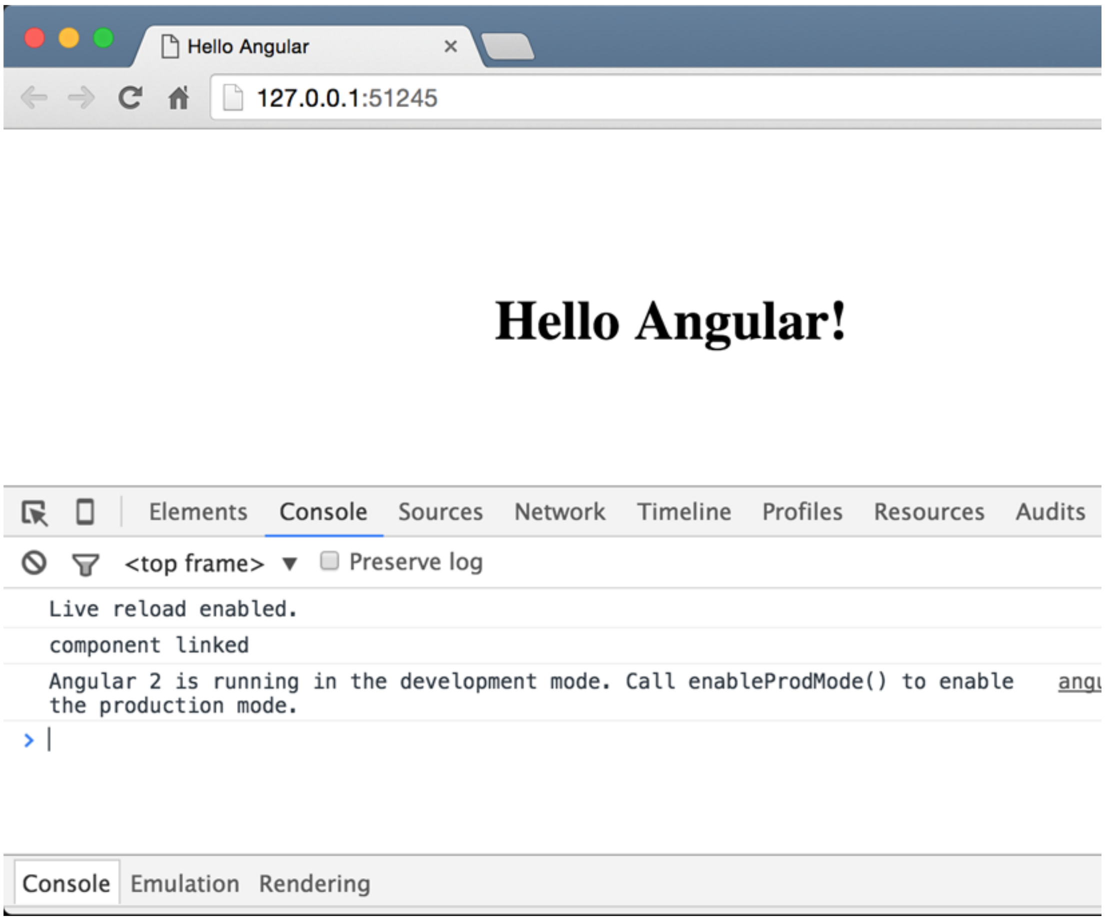
Now we can use our component in the body of the html:

```
<body>
  <app>Loading ...</app>
</body>
```

It is finally time to serve the app. You can serve the app in the current directory using the `live-server`:

```
live-server .
```

If everything is wired up correctly, you should be able to see the following:



```
Live reload enabled.
component linked
Angular 2 is running in the development mode. Call enableProdMode() to enable    angu
the production mode.
```

Running a basic component in the browser

**Debugging the component**

You can connect chrome's debugger to VSCode using the chrome debugger extension for Visual Studio Code. See t App from VSCode section in case you missed to install it. But, assuming that you have the extension installed, you ca app from VSCode. In order to do that, we need to create a `launch.json` file in the `.vscode` folder:

```
touch .vscode/launch.json
```

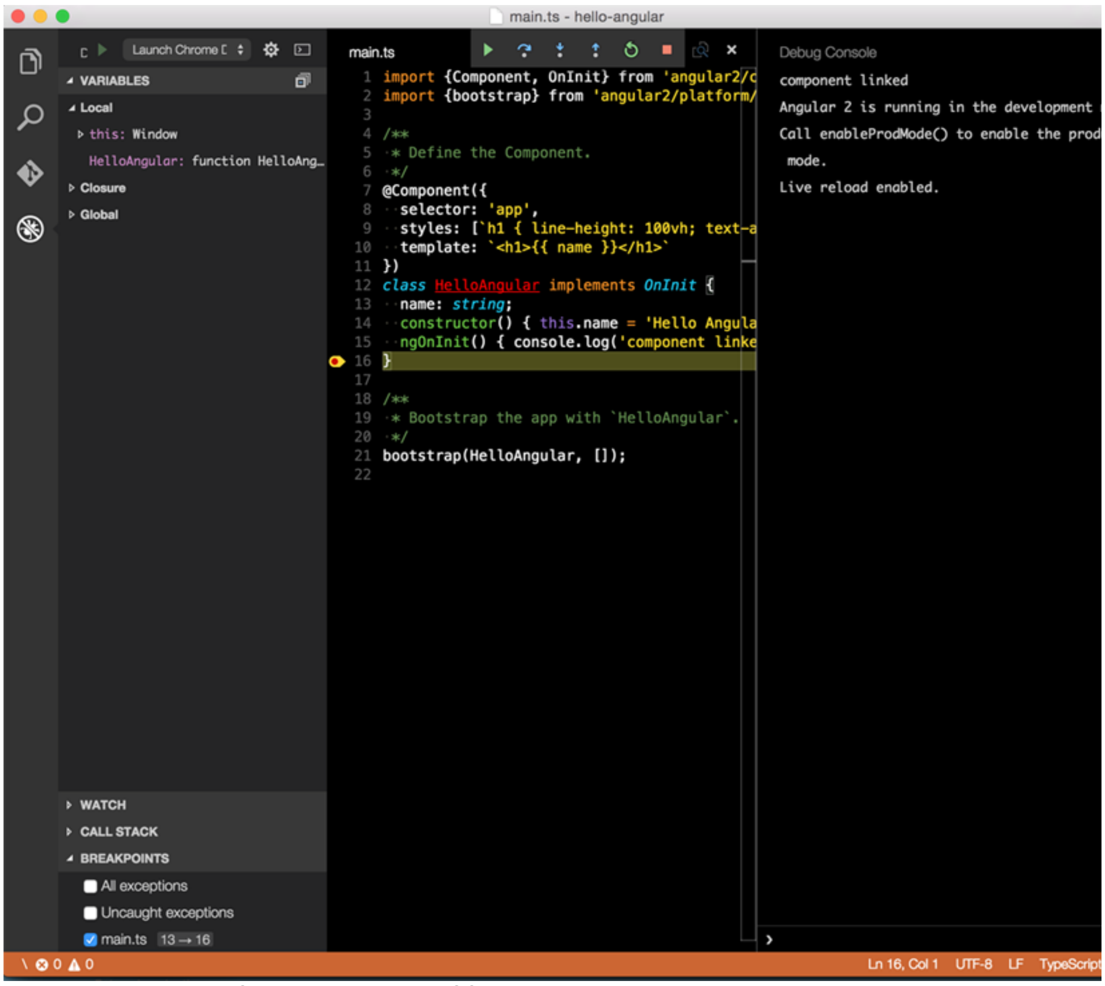
After you created the file, put in the following configuration in the file:

```
{
  "version": "0.1.0",
  "configurations": [
    {
      "name": "Launch Chrome Debugger",
      "type": "chrome",
      "request": "launch",
      "url": "http://127.0.0.1:8080/",
      "sourceMaps": true,
      "webRoot": ".",
      "runtimeExecutable": "/Applications/Google Chrome.app/Contents/MacOS/Google Chrome",
      "runtimeArgs": [
        "--remote-debugging-port=9222",
        "--incognito"
      ]
    }
  ]
}
```

Before running the debugger:

- Make sure that all instances of chrome are closed. It makes it easier to run the debugger from VSCode itself.
- Make sure that the `runtimeExecutable` path is valid. This value would be different depending on your OS.
- Make sure that the `url` value is valid as well. The `url` value has to match the path that you see when you run a the files.
- Set a breakpoint on a line in `main.ts` file and then run the debugger under the debugger tab.

In order to run the debugger, select `Launch Chrome Debugger` in the dropdown under the debugger tab and either c icon or hit F5 on the keyboard. After that, an instance of Chrome should be opened in incognito mode. In order to trig debugger just refresh the page and you should be able to see the debugger pausing in VSCode. If everything is set u should be able to see something like the following screenshot:



```
import {Component, OnInit} from 'angular2/c
import {bootstrap} from 'angular2/platform/

/**
 * Define the Component.
 */
@Component({
  selector: 'app',
  styles: [`h1 { line-height: 100vh; text-a
  template: `<h1>{{ name }}</h1>`
})
class HelloAngular implements OnInit {
  name: string;
  constructor() { this.name = 'Hello Angula
  ngOnInit() { console.log('component linke
}

/**
 * Bootstrap the app with `HelloAngular`.
 */
bootstrap(HelloAngular, []);
```

```
component linked
Angular 2 is running in the development
Call enableProdMode() to enable the prod
 mode.
Live reload enabled.
```

Debugging the app with Chrome Debugger in VSCode

## Component Inputs

- You can pass data to a component.
- You can either use the `inputs` array on a component or annotate an instance variable with the `Input` decorator
- Once you specify the inputs to your component, they become available in the `ngOnInit` method
- You can implement the `ngOnInit` and access the input instance variables
- You can use the `[propname]="data"` to set the `propname` to whatever `data` evaluates to
- Note that if you set `[propname]="'data'"` , `propname` will be set to the literal `data` string

### Project files

The project files for this section are in angular2-intro/project-files/angular-examples/component-input.

### Getting Started

In order to demonstrate component inputs, we are going to create a `user` component and pass `name` , `lastName` to it. So our final html tag would look something like the following:

```
<user name="Tom" lastName="Johnson" uesrId="1"></user>
```

And the template for the component will be:

```
<h1>Hello, {{ name }} {{ lastName }}, id: {{ userId }}</h1>
```

which would output: `Hello, Tom Johnson id: 1`.

To get started, let's define the `User` component:

```
1 @Component({
2   selector: 'user',
3   template: '<h1>Hello, {{ name }} {{ lastName }} id: {{ userId }}</h1>',
4   inputs: ['name', 'lastName', 'userId'] // <- specifying the inputs to the `User` component
5 })
6 class User {}
```

- On line 4 we are defining the inputs as an array of strings

Then, we are going to use the `User` component inside our app's template:

```
@Component({
  selector: 'app',
  template: `<user name="Tom" lastName="Johnson" uesrId="1"></user>`
})
class Root {}
```

because we are using the `User` component in the app, we need to register it with the app by adding `User` class t
`directives` of the app component:

```
@Component({
  selector: 'app',
  template: `<user name="Tom" lastName="Johnson" userId="1"></user>`,
  directives: [User] // <- register the component
})
class Root {}
```

and at the end we need to bootstrap the app:

```
bootstrap(Root, [])
```

Now, notice that instead of adding the inputs to the `inputs` array, we could have decorated the instance variables
decorator:

```
import {Input} from 'angular2/core'; // <- importing the Input decorator
@Component({
  selector: 'user',
  template: '<h1>Hello, {{ name }} {{ lastName }} id: {{ userId }}</h1>'
  // <- removing the inputs array.
})
class User {
  @Input() private name: string;
  @Input() private lastName: string;
  @Input() private userId: number;
}
```

### Binding Data to Properties

Now, let's see how we can bind to a property from another component. For this example, we are going to continue w
component and create a new component called `Permission`. Then we are going to use the the `Permission` comp
`User` component and set the `uid` of `Permission` by the `userId` of the `User`.

The `Permission` component is defined as follows:

```
1  @Component({
2    selector: 'permission',
3    template: '<h2> Restriction is: {{ restriction }}'
4  })
5  class Permission {
6    @Input() private uid: string;
7    private restriction: string;
8    constructor() {
9      this.restriction = 'none';
10   }
11   ngOnInit() {
12     this.restriction = this.uid === '1' ? 'admin' : 'normal';
13   }
14 }
```

- On line 6 we are defining `uid` to be an input instance variable. It's value is set from outside.
- In the constructor we are setting a default value for the restriction.
- Then in the `ngOnInit` hook, we are evaluating the value of `restriction` based on the given id provided by other
this case the `User` component
- In this silly example, if the passed id is `1`, we will set the `restriction` to `admin`, otherwise we set it to `normal`

then we are going to register the `Permission` component with the `User` component so that we can use it in the `U`

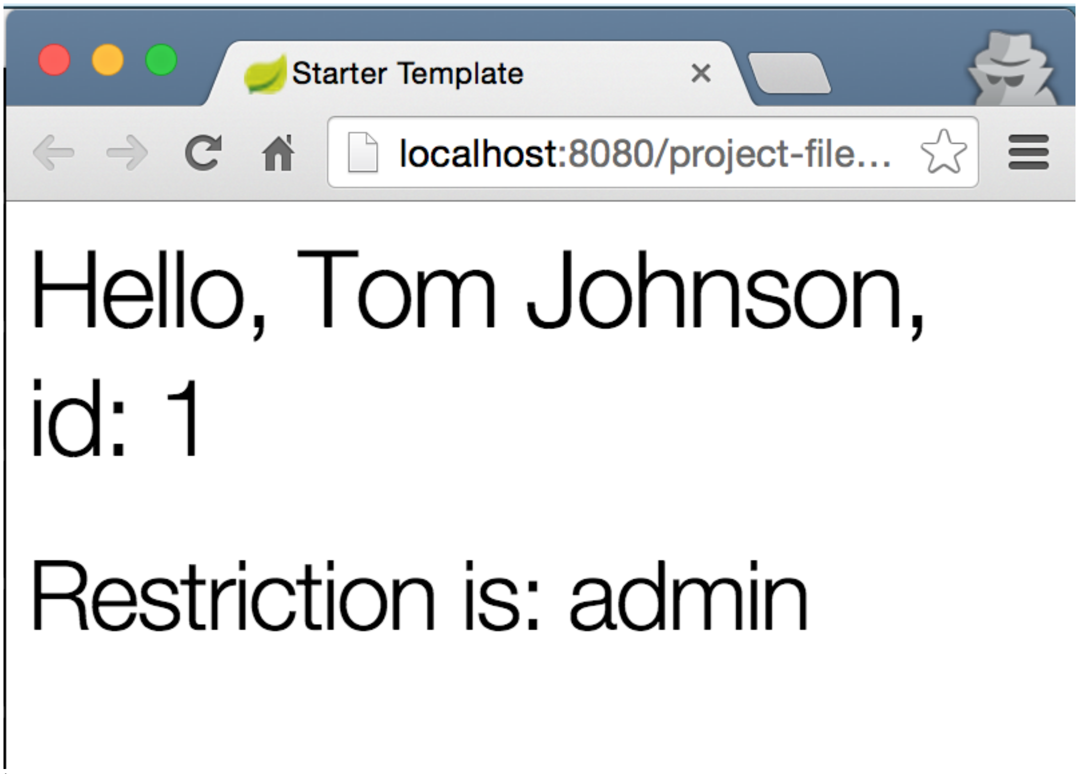
```
@Component({
  selector: 'user',
  ///...
  directives: [Permission] // <-
})
class User {}
```

then we can update the `User` template to include the `Permission`:

```
@Component({
  selector: 'user',
  template: `
  <h1>Hello, {{ name }} {{ lastName }}, id: {{ userId}}</h1>
  <div>
    <permission [uid]="userId"></permission>
  </div>
  `,
  inputs: ['name', 'lastName', 'userId'],
  directives: [Permission]
})
class User {}
```

- Note that on line 6 we are setting the `uid` of `Permission` by `userId` available from the `User` component.

If you run the app you should see the following printed to the page:



Hello, Tom Johnson, id: 1

Restriction is: admin

Input to components

## Binding to DOM Properties

In addition to custom properties, you can bind to DOM properties. Below are some examples:

**Binding to `style`**

We can bind to the style property of DOM nodes. In the example below, if the value of `isDone` is true, we set the `style.textDecoration` to `line-through`, otherwise we won't set it to anything

```
<div [style.textDecoration]="isDone ? 'line-through' : ''"> Todo Item </div>
```

**Binding to `class`**

In addition to the `style` property, we can also bind to the `class` property. In the example below, we are setting th
"collapsed" if the `isCollapsed` value is true and `expanded` if the the value is false:

```
<div [className]="isCollapsed ? 'collapsed' : 'expanded'">Element</div>
```

**Binding to 'hidden'**

You can bind to the hidden property of a DOM node and show or hide the element. In the example below, we are hid
if the `isVisible` value is true:

```
<div [hidden]="isVisible">To Hide element</div>
```

**Binding to `textContent`**

We can bind to the `textContent` property and set the text content of a node. In the example below, we are setting t by reading the value of an input:

```html
<div [textContent]="textValue"></div>
<input type="text" #contentInput>
<button (click)="setTextContent()">Set Text Content</button>
```

## Component Output/Events

- Events can be emitted from components. These events can be either custom or they could be DOM events
- The syntax is `(eventname)="fn()"` where `eventname` is the name of the event and `fn` is the handler function
- The handler function is called when the event is fired
- For example, if you want to handle a click event you can do: `(click)="handler()"`. In this case the `hander` is ca
the click event is fired off
- You can use Angular's `EventEmitter` to fire off custom events

### Custom Output Events

**Project Files**

The project files for this section are in angular2-intro/project-files/angular-examples/component-output-events.

**Final Result**

The goal of this section is to show you how to create a component that contains a button that when is clicked, calls a
by the component's class. The final html will look like the following:

```html
<p>Value: {{ value }}</p>
<button (click)=addOne()>Add +</button>
```

That idea is very simple: every time we click on the button we want to increment the value by one. In addition to that,
able to hook into a custom event and run the `addOne` method whenever the event is fired:

```html
<p>Value: {{ value }}</p>
<span adder-auto (myevent)=addOne()>adding ...</span>
```

**Getting Started**

Let's get started by defining our `Adder` component:

```typescript
@Component({
  selector: 'adder',
  template:`
  <p>Value: {{ value }}</p>
  <button (click)=addOne()>Add +</button>
  `
})
class Adder {
  private value: number;
  constructor() {
    this.value = 0;
  }
  addOne() {
  this.value += 1;
  console.log(this.value);
  }
}
```

Now, we are just going to register `Adder` with our root component:

```
@Component({
  selector: 'app',
  directives: [Adder],
  template: '<adder></adder>'
})
class App {}
```

after you bootstrap the app and run it you should be able to see a button that when clicked increments the value by c

**Using EventEmitter**

Now, let's see how we can use the `EventEmitter` to increment the value by one every time a custom event is fired e order to achieve that, we are going to create an attribute directive called `AdderAuto`. Start by importing the `Directi` class:

```
import {Directive} from 'angular2/core';
```

and then define the selector for the directive:

```
@Directive({
  selector: '[adder-auto]'
})
```

- `selector: '[adder-auto]'` means that angular will target any element that has the `adder-auto` attribute and wi instance of the class. Now we need to define the class for our directive:

```
class AdderAuto {
  // custom event definition
}
```

In this class we need to define a custom event output hook. We are going to call it `myevent`. The same way that you `(click)`, we want to be able to use `(myevent)`. To achieve that, we need to create an instance variable and deco `Output` decorator:

```
// -> importing `EventEmitter` and `Output` decorator.
import {EventEmitter, Output} from 'angular2/core';
class AdderAuto {
  @Output() myevent: EventEmitter<string>;
  constructor() {
    this.myevent = new EventEmitter();
  }
}
```

- If you notice, `myevent` is of type `EventEmitter` that emit events of type string
- In the constructor we are creating an instance of `EventEmitter`. So now we can use `myevent` to emit events
- We can use `setInterval` to emit event from our custom event every second

```
class AdderAuto {
  @Output() myevent: EventEmitter<string>;
  constructor() {
    this.myevent = new EventEmitter();
    setInterval(()=> {this.myevent.emit('myevename')}, 1000);
  }
}
```

Now we can register `AdderAuto` with the `Adder` component and run the `addOne` method every second:

```
@Component({
  selector: 'adder',
  ...
  directives: [AdderAuto] // <- register `AdderAuto`
})
```

and then we can update the template:

```
<p>Value: {{ value }}</p>
<button (click)="addOne()">Add +</button>
<!-- using the event. -->
<h2>Using Emitter</h2>
<span adder-auto (myevent)="addOne($event)"> EVENT: </span>
```

- first we are adding the attribute directive `adder-auto` on the span
- second, we are using the `myevent` hook and attaching `addOne` handler to it. This means that whenever the `myev`
triggered, run the `addOne` handler.

The `Adder` component now looks like the following with the updated template:

```
@Component({
  selector: 'adder',
  template:`
  <p>Value: {{ value }}</p>
  <button (click)="addOne()">Add +</button>
  <h2>Using Emitter</h2>
  <span adder-auto (myevent)="addOne($event)"> EVENT: </span>
  `,
  directives: [AdderAuto]
})
```

Now if you run the code, you should be able to see the number incrementing by one every second.

## Binding to DOM Events

In addition to custom output events, we can bind to almost all of the native DOM events. Below you can find some co
with examples:

**Inputs**

- keyup

```
<input type="text" #inputField2 (keyup)="setInputFieldValue2(inputField2.value)">
<pre> {{ keyupValue }}</pre>
```

- input

```html
<input type="text" #inputField (input)="setInputFieldValue(inputField.value)">
<pre> {{ inputValue }}</pre>
```

### Mouse

- click

```html
<button (click)="sayHello()"> Say Hello! </button>
<pre>{{ hello }}</pre>
```

- dblclick

```html
<button (dblclick)="sayDoubleHello()"> Say Hello! </button>
<pre>{{ doubleHello }}</pre>
```

- mousedown

```html
<button (mousedown)="sayDownHello()">Say Hello !</button>
<pre>{{ downHello }}</pre>
```

- mouseup

```html
<button (mouseup)="sayUpHello()">Say Hello !</button>
<pre>{{ upHello }}</pre>
```

- mouseenter

```html
<h2 (mouseenter)="sayEnterHello()" (mouseleave)="clearEnterHello()">Mouseenter/leave</h2>
<pre>{{ enterHello }}</pre>
```

- mouseleave

```html
<h2 (mouseenter)="sayEnterHello()" (mouseleave)="clearEnterHello()">Mouseenter/leave</h2>
<pre>{{ enterHello }}</pre>
```

## Event Delegation/Bubbling

When an event is raised by an element, by default it bubbles up the html hierarchy. For example, if you have a table a
click handler to the table itself, you can catch the row that was clicked by only attaching a single handler. This metho
you are dealing with situations where you don't want to attach event handlers to every elements, and you just want tc
Below is an example of event delegation, detecting the row that was clicked on the table:

```html
<pre>{{rowClicked}}</pre>

<table (click)="delegate($event)">
  <tr>
    <td>1</td>
  </tr>
  <tr>
    <td>2</td>
  </tr>
  <tr>
    <td>3</td>
  </tr>
  <tr>
    <td>4</td>
  </tr>
</table>
```

Notice how we have a single `click` handler only on the table itself. The `$event` is the bubbled event that we can c
interesting stuff with. In this case we are just reading the `textContent` from the target: `event.target.textContent`

```
catchBubbledEvent(event) {
  this.rowClicked = event.target.nodeName === 'TD' ? event.target.textContent : '';
}
```

In the method above we are checking if a `td` was clicked on. If so, we set the `this.rowClicked` to the td's value,
it to an empty string.

## ViewChildren

- The children elements located inside of its template of a component are called
- Metadata classes: `@ViewChildren` , `@ViewChild`

**TODO**

## ContentChildren

- Elements used between the opening and closing tags of the host element of a given component
- Metadata classes: `@ContentChildren` , `@ContentChild`

**TODO**

## ViewProviders

**TODO**

## Providers

**TODO**

# Directives

- Directives and components hand-in-hand are the fundamental building blocks of any Angular app
- Directives are components without templates. Conversely, components are directives without templates
- Directives allow you to attach behavior to elements in the DOM
- A directive is defined using the `@directive` decorator
- There are three types of directives in Angular:
  - Structural
  - Attribute
  - Components
- Evey directive metadata, has the following options:

    - selector
    - host
    - ...
- The `selector` attribute uses a css selector to find the element. However, parent-child relationship selectors are n
- You can use the following possible selectors:
    - `element`
    - `[attribute]`
    - `.classname`
    - `:not()`
    - `.some-class:not(div)`
- The `host` option defines:
    - Property bindings
    - Event handlers
    - attributes

**TODO**(other decorator options)

## Web Components Basics

Web Components are made up four specifications:

- Custom Elements: enabling custom html tags
- Shadow DOM: enabling isolation for custom elements
- HTML Templates: enabling to define html template fragments
- HTML Imports: enabling html fragment imports

#### Custom Elements

**TODO**

#### Shadow DOM

- Enables a node to express three subtrees:
    - Light DOM: visible DOM notes inside the custom element tag/DOM supplied by the consumer
    - Shadow DOM: private to the element and hidden from others and attached to the element's shadow root
    - Composed DOM: Rendered by the browser by distributing the light DOM into the Shadow DOM
    - *Logical DOM* = Light DOM + Shadow DOM. The developer interacts with this layer

**TODO**

#### HTML Templates

**TODO**

#### HTML Imports

**TODO**

## Attribute Directives

- The Attribute directive changes the appearance or behavior of an element.
- Angular has several built-in attribute directives, namely `NgClass` and `NgStyle`
- It is a good idea to prefix your directives with a prefix. You cannot use the `ng` prefix since it's already used by Ang

- When you apply the attribute directive to an element, the element will be knownn as the **host**.
- For example, if you had a directive called `my-directive` and applied it in `<div class="hello"> <span my-directive> ... </span> </div>`, the `span` would be the **host**.

**TODO** (writing a custom attribute directive)

```
@Directive({
  selector: '[simple-directive]',
  host: {
    '(mouseleave)': 'onMouseLeave()',
    '(click)': 'onClick()',
    '[hidden]': 'isHidden',
    '[class.done]': 'isDone',
    'role': 'button'
  }
})
class SimpleDirective implements OnInit {
  @Input() private color: string
  @Output() myevent: EventEmitter<string>;
  private isHidden: boolean = false;
  private isDone: boolean = false;
  private defaultColor:string = 'magenta';
  private elm: any;

  constructor(private elmRef: ElementRef, private renderer: Renderer) {
    this.elm = elmRef.nativeElement;
    this.myevent = new EventEmitter();
    setInterval(() => {this.myevent.emit('myevename')}, 1000);
  }
  ngOnInit() {
    this.defaultColor = this.color || this.defaultColor;
    this.setColor(this.color || this.defaultColor);
  }
  private setColor(color: string) {
    this.renderer.setElementStyle(this.elm, 'color', color);
  }
  set setIsHidden(state) { this.isHidden = state; }
  set setIsDone(state) { this.isDone = state; }

  onMouseLeave() { this.setColor(this.defaultColor); }
  onClick() { this.setColor('orange') }
}
```

**selector** TODO: details

**host** TODO: details

**Input** TODO: details

**Output** TODO: details

**ElementRef** TODO: details**

**Renderer** TODO: details**

## Structural Directives

- The Structural directive changes the DOM layout by adding and removing DOM elements
- Angular has several built-in structural directives, namely `NgIf`, `NgSwitch`, and `NgFor`
- When working with structural directives, we should ask ourselves to think carefully about the consequences of addi

removing elements and of creating and destroying components

- Angular uses the html5 `<template>` tag to add or remove DOM elements
- By default, Angular replaces `<template>` with `<script>` tag if no behavior is attached
- The `*` before a directive name is a shorthand for including the directive content in the `<template>` tag
- Below you can see the built-in `NgIf` directive with and without the asterisks `*` :

**With `*`**

```
<p *ngIf="condition"></p>
```

**Without `*`**

```
<template [ngIf]="condition">
  <p></p>
</template>
```

Notice how the `<p>` tag is wrapped with a `<template>` and the condition is bound to the `[ngIf]` property of the

**TODO** (writing a custom structural directive)

```
@Directive({
  selector: '[myUnless]'
})
class UnlessDirective {

  constructor(
    private tRef: TemplateRef,
    private vContainer: ViewContainerRef
  ) { }

  @Input() set myUnless(condition: boolean) {
    if (!condition) {
      this.vContainer.createEmbeddedView(this.tRef);
    } else {
      this.vContainer.clear();
    }
  }
}
```

**TemplateRef**: TODO: details

**ViewContainerRef**: TODO: details

**`@Input() set myUnless(condition: boolean) {}`** : TODO: details

## Built-in Directives

Angular has a couple of useful built-in directives.

**TODO**(Note on directive names, docs and template usage)

### `NgClass`

- `import {NgClass} from 'angular2/common';` , `directives: [NgClass]`
- Template Usage: `<div class="button" [ngClass]="{active: isActive, disabled: !isActive}"`

**Note** that we are using `ngClass` in the template, but not `NgClass`

`NgIf`

**Usage**

```html
<div *ngIf="isDone">{{ list }}</div>
```

or in long-hand form:

```html
<template>
  <div [ngIf]="isDone">{{ list }}</div>
</template>
```

**Details**

From the docs: "The ngIf directive does not hide the element. Using browser developer tools we can see that, when t
true, the top paragraph is in the DOM and the bottom disused paragraph is completely absent from the DOM! In its p
`<script>` tags. We could hide the unwanted paragraph by setting its css display style to none. The element would r
DOM while invisible. Instead we removed it with ngIf.

The difference matters. When we hide an element, the component's behavior continues. It remains attached to its DC
continues to listen to events. Angular keeps checking for changes that could affect data bindings. Whatever the comp
doing it keeps doing.

Although invisible, the component — and all of its descendant components — tie up resources that might be more us
The performance and memory burden can be substantial and the user may not benefit at all.

On the positive side, showing the element again is very quick. The component's previous state is preserved and read
component doesn't re-initialize — an operation that could be expensive.

ngIf is different. Setting ngIf to false does affect the component's resource consumption. Angular removes the elemer
stops change detection for the associated component, detaches it from DOM events (the attachments that it made) a
component. The component can be garbage-collected (we hope) and free up memory.

Components often have child components which themselves have children. All of them are destroyed when ngIf destı
common ancestor. This cleanup effort is usually a good thing.

Of course it isn't always a good thing. It might be a bad thing if we need that particular component again soon.

The component's state might be expensive to re-construct. When ngIf becomes true again, Angular recreates the cor
subtree. Angular runs every component's initialization logic again. That could be expensive ... as when a component ı
that had been in memory just moments ago."

`NgSwitch`

**Usage**

```html
<div [ngSwitch]="status">
  <template [ngSwitchWhen]="'inProgress'">In Progress</template>
  <template [ngSwitchWhen]="'isDone'">Finished</template>
  <template ngSwitchDefault>Unknown</template>
</div>
```

**TODO**

`NgFor`

**Usage**

```
<ul>
  <li *ngFor="#item of items">{{ item }}</li>
</ul>
```

or in long-hand form:

```
<ul>
  <template ngFor #item [ngForOf]="items">
    <li>{{ item }}</li>
  </template>
</ul>
```

**TODO**(Details)

### Accessing Directives from Parents

**TODO** (access directives on parent elements)

### Accessing Directives from Children

**TODO** (access directives on children and descendants)

## Change Detection

- In Angular2 you can limit the change detection scope to components
- Using `chageDection` property we can choose a change detection strategy for a component
- The `changeDetection` field accept one of the following values:
  - `ChangeDetectionStrategy.Default` : sets detector mode to `CheckAlways`
  - `ChangeDetectionStrategy.OnPush` : sets detector mode to `CheckOnce` . This will limit change detection to the bindings af component only
  - `ChangeDetectionStrategy.Detached` : change detector sub tree is not a part of the main tree and should be skipped
  - `ChangeDetectionStrategy.CheckAlways` : after calling detectChanges the mode of the change detector will remain `Check`
  - `ChangeDetectionStrategy.Checked` : change detector should be skipped until its mode changes to `CheckOnce`
  - `ChangeDetectionStrategy.CheckOnce` : after calling detectChanges the mode of the change detector will become `Checke`
- Having the ability to specify change detection strategy can reduce the number of checks and improve app's perforr

## Pipes

- Pipes allow you to transform values in templates before they are outputed to the view.
- Pipes were formerly known as filters in Angular 1.x
- A pipe is defined using the `@pipe` class decorator
- The pipe decorator takes name as a parameter defining the name of the pipe: `@pipe({ name: 'myPipe' })`
- Every pipe class has a `transform` method that transforms input to outputs:
  - The first parameter is the input to the pipe
  - The second parameter is the list of arguments passed to the pipe
- Give the following pipe in a template: `{{ data | somePipe:1:'px'}}` :
  - `data` is the input to pipe -- the first parameter of the transform method
  - `[1, 'px']` is the arguments to the pipe -- the second parameter of the transform method

- A pipe can be as simple as:

```
@pipe({name: 'simplePipe'})
class MyPipe {
  transform(input, args) { return input + 'px'; }
}
```

- If you want to use a pipe, you need to register your pipe class with the components in the pipes array:

```
@component({
  selector: '...',
  pipes: [MyPipe] // adding pipe to the array of pipes.
})
class MyComponent {}
```

- Pipes can be chained: `input | pipe1 | pipe2 | pipe3`
  - `input | pipe1 : output1`
  - `output1 | pipe2: output2`
  - `output2 | pipe3 : finalOutput`

## Basic Pipe

Let's make a basic pipe called `pixel` that takes a value as the input and appends 'px' to the end of it. The project fi
section are in angular2-intro/project-files/angular-examples/pipes/basic-pipe.

Start by making a copy of the "starter" folder and call it "basic-pipe" and put it in `project-files/angular-examples`
folder in VSCode: `code project-files/angular-examples/basic-pipe` and start the build with `command + shift +`

Then, create a file for the pipe and call it `pixel.pipe.ts` in the root of the project.

After that we need to do couple of things to define the pipe:

- Import the Pipe Class Metadata from angular core: `import {Pipe} from 'Angular/core'`
- Then create a class defining the Pipe:

```
class PixelPipe {

}
```

- Implement the `transform` method in the class:

```
class PixelPipe {
  transform(input) {
    return input + 'px';
  }
}
```

- After implementing the method, we need to decorate the class and give the pipe a name that we want to use in our

```
@Pipe({name: 'pixel'}) // <- adding the decorator
class PixelPipe {
  transform(input) {
    return input + 'px';
  }
}
```

- As the last step we are going to export the class by putting the `export` keyword behind the class:

```
...
export class PixelPipe {
  ...
}
```

Now, your file should look like the following:

```
import {Pipe} from 'angular2/core';
@Pipe({name: 'pixel'}) // <- adding the decorator
export class PixelPipe {
  transform(input) {
    return input + 'px';
  }
}
```

Now, let's go back to the `main.ts` file and import our pipe:

```
import {Component} from 'angular2/core';
import {bootstrap} from 'angular2/platform/browser';
import {PixelPipe} from './pixel.pipe'; // <- importing pipe
```

After importing our pipe, we should register it with our component by adding it to the `pipes` array:

```
@Component({
  selector: 'app',
  templateUrl : 'templates/app.tpl.html',
  pipes: [PixelPipe] // <- registering the pipe
})
```

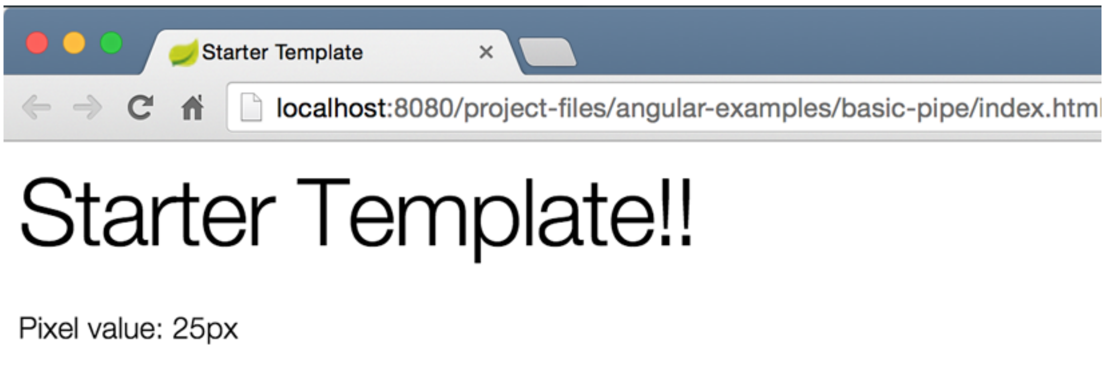
Now that we have registered the pipe, we can use it in our template in `templates/app.tpl.html` :

```
<h1>{{ name }}</h1>
<p>Pixel value: {{ 25 | pixel }}</p>
```

You should be all set now. You can set the url in your `launch.json` file and hit F5:

```
...
"url": "http://localhost:8080/project-files/angular-examples/basic-pipe/index.html",
...
```

If your server is running you should be able to see the following output:


Starter Template
localhost:8080/project-files/angular-examples/basic-pipe/index.html

Starter Template!!

Pixel value: 25px

Running the pixelPipe in the browser

## Chaining Pipes

Let's continue where we left off with the "pixelPipe" and add another pipe called "round" that rounds down given valu

```
25.3 | round | pixel -> 25px
```

The project files for this section are in angular2-intro/project-files/angular-examples/pipes/pipe-chaining.

We are going to add the "roundPipe" to our "basic-pipe" project. Let's get started by adding the `round.pipe.ts` file the project:

```
import {Pipe} from 'angular2/core';
@Pipe({name: 'round'})
export class RoundPipe {
  transform (input) {
    return Math.floor(+input); // <- convert input to number and then floor it.
  }
}
```

This Pipe is not complicated at all. We are just returning the floor of the input. We are also converting the input to nun `+` before input.

Now, let's import the pipe into our `main.ts` file:

```
import {Component} from 'angular2/core';
import {bootstrap} from 'angular2/platform/browser';
import {PixelPipe} from './pixel.pipe';
import {RoundPipe} from './round.pipe'; // <- importing `RoundPipe`
```

and then we have to add the pipe to the list of pipe array:

```
@Component({
  selector: 'app',
  templateUrl : 'templates/app.tpl.html',
  pipes: [PixelPipe, RoundPipe] // <- registering the pipe
})
```

after that we are going to add the following to our `templates/app.tpl.html` file:

```
<p>Pixel value: {{ 34.4 | round | pixel }}</p>
```

After running the app you should see `34.px` as the output on the page.

## Pipes with Parameters

In this section we are going to extend our 'pixel' pipe to accept an optional parameter to set the unit. As a result, we
rename the 'pixel' pipe to 'unit' to make it more generic. This pipe will take the unit as an optional argument. If no arg
passed, it will default to 'px'. That is:

```
25 | unit -> 25px
25 | unit:'em' -> 25em
34.5 | round | unit:'%' -> 34%
```

You can look at the project files in angular2-intro/project-files/angular-examples/pipes/pipe-unit.. AFter refactoring the
Pipe, we just need to change the implementation of the "UnitPipe":

**`unit.pipe.ts`**

```
1  import {Pipe} from 'angular2/core';
2  @Pipe({name: 'unit'})
3  export class UnitPipe {
4    transform(input, args:string) {
5      const unit = args[0] || 'px';
6      return input + unit;
7    }
8  }
```

- On line 5, we are grabbing the first parameter that is passed in and setting it to the `unit` variable. And if the value
are setting 'px' as the default value.
- And finally we are returning `input + unit`.

That's basically all we have to do. Note that you can pass multiple parameters separated by `:` and they all become
`args` array. So if you wanted to expand this pipe, this is how your template would look like:

```
{{ 25 | unit:'em':2}}
```

And the `args` array would be: `['em', 2]`.

## Async Pipes

Async Pipes can be used for values that will be resolved after some asynchronous operation like getting a value after
call.

**TODO**

## Built-in Pipes

In this section we are going to look at the pipes that Angular provides out of the box.

- AsyncPipe: used to work with asynchronous values
- CurrencyPipe: used to format a number as a local currency
- DatePipe: used to format a date object to a readable string
- DecimalPipe: used to format numbers
- JsonPipe: calls `JSON.stringify` on the input and useful for debugging
- LowerCasePipe: used to convert a string to lowercase letters

- PercentPipe: used to format a number as percentage
- SlicePipe: used to create a subset of list or string
- UpperCasePipe: used to transform a text to upper case

## Date

```
{{ input | date:optionalFormat}}
```

- `input` : a date object or a number (milliseconds since UTC epoch)
- `optionalFormat` : a string used to format the output. It specifies which components of date and time to include in

**Using predefined formats**

Usage Example: `{{ input | date:'short'}}`

| Name | Example |
|---|---|
| `short` | 9/3/2010, 12:05 PM |
| `shortDate` | 9/3/2010 |
| `medium` | Sep 3, 2010, 12:05:08 PM |
| `mediumDate` | Sep 3, 2010 |
| `longDate` | September 3, 2010 |
| `fullDate` | Friday, September 3, 2010 |
| `shortTime` | 12:05 PM |
| `mediumTime` | 12:05:08 PM |

**Using Custom Formats**

- Generally speaking every date object has a year, month, day, hour, minute, and second.
- Using a custom string, you can specify which component you would like to include in the output.

| Year | Month | Day | Weekday | Hour | Minute |
|---|---|---|---|---|---|
| y | M, MMM, MMMM | d | EEE, EEEE | j, h, H | m |
| 2016 | 1, Jan, January | 1 | Sun, Sunday | 1, 1 AM, 1 | 1 |

Note that every single letter identifier can be used twice to denote a double digit numeric value. For example, `yy` wil
for the year value. Below is a table just to be thorough:

**Double Digit**

| Year | Month | Day | Hour | Minute | Se |
|---|---|---|---|---|---|
| yy | MM | dd | jj, hh, HH | mm | ss |
| 16 | 01 | 01 | 01, 01 AM, 01 | 01 | 01 |

Details for Month/Weekday/Hour are summarized in the tables below:

**Month Details**

| M | MMM | MMMM |
|---|---|---|
| Digit | Abbr Name | Full Name |
| 1 | Jan | January |

**Weekday Details**

| EEE | EEEE |
|---|---|
| Abbr Name | Full Name |
| Sun | Sunday |

**Hour Details**

| j | h | H |
|---|---|---|
| Digit | Hour12 AM/PM | Military |
| 13 | 1 PM | 13 |

## Slice

- The slice pipe is useful when you want a subset of a list or string. One of the common use cases are in when iterati with `ngFor` for example.

**TODO**

## Decimal

- Used for formatting numbers given a decimal formatter

**TODO**

## Percent

**TODO**

### Currency

**TODO**

### Async

**TODO**

### Json

**TODO**

### LowerCase

**TODO**

### UpperCase

**TODO**

# Dependency Injection

Dependency Injection is a coding pattern in which a class receives its dependencies from external sources rather thar itself. In order to achieve Dependency Injection we need a Dependency InjectionFramework to handle the dependenc a DI framework, you simply ask for a class from the injector instead of worrying about the dependencies inside the cla

Angular has a standalone module that handles Dependency Injection. This framework can also be used in non-Angula handle Dependency Injection.

**TODO**

# Services and Providers

- A service is nothing more than a class in Angular 2. It remains nothing more than a class until we register it with the injector.
- When you bootstrap your app, Angular creates an injector on the fly that can inject services and other dependencie app.
- You can register the service or the dependencies during when bootstrapping the app or when defining a componer
- If you have a class called `MyService` , you can register it with the Injector and then you can inject it everywhere:

```
bootstrap(App, [MyService]); // second param is an array of providers
```

- Providers is a way to specify what services are available inside the component in a hierarchical fashion.
- A provider can be a class, a value or a factory.
- Providers create the instances of the things that we ask the injector to inject.
- `[SomeService];` is short for `[provide(SomeService, {useClass:SomeService})];` where the first param is the to second is the definition object.
- A simple object can be passed to the Injector to create a Value Provider:

```
beforeEachProviders(() => {
  let someService = { getData: () => [] };
  // using `useValue` instead of `useClass`
  return [ provide(SomeSvc, {useValue: someService}) ];
});
```

- You can also use a factory as a provider.
- You can use a factory function that creates a properly configured Service:

```
let myServiceFactory = (dx: DepX, dy: DepY) => {
  return new MyService(dx, dy.value);
}

// provider definition object.
let myServiceDefinition = {
   useFactory: myServiceFactory,
   deps: [DepX, DepY]
};

// create provider and bootstrap
let myServiceProvider = provide(MyService, myServiceDefinition);
bootstrap(AppComponent, [myServiceProvider, DepX, DepY]);
```

- Defining object dependencies is simple. You can make a plain JavaScript object available for injection using a string and the `@Inject` decorator:

```
var myObj = {};

bootstrap(AppComponent, [
  provide('coolObjToken', {useValue: myObj})
]);

// and you can inject it to a component

import {Inject} from 'angular2/core'
constructor(dx: DepX, @Inject('coolObjToken') config)
```

## Simple Service

In this section we are going to make a simple service and use it in our root component.

**Project Files**

The project files for this section are in angular2-intro/project-files/angular-examples/services/simple-service;

**Getting Started**

Let's get started by creating a class, called `StudentSvc` that represents our service:

```
class StudentSvc {
  private students: any[];
  constructor() {
    this.students = [
      {name: 'Tom', id: 1},
      {name: 'John', id: 2},
      {name: 'Kim', id: 3},
      {name: 'Liz', id: 4}
    ];
  }
  getAll() {
    return this.students;
  }
}
```

There is nothing special about this class. It's just a class the has a method to return the list of all students. Now, we a make it special by decorating it with the `Injectable` decorator. But, first we need to import `Injectable` from Angu

```
import {Injectable} from 'angular2/core';
```

After importing the `Injectable` metadata class, we can decorate our class:

```
/**
 * Student service
 */
@Injectable() // <- decorating with `Injectable`
class StudentSvc {
  private students: any[];
  constructor() {
   // ...
  }
  // ...
}
```

Now we have an injectable class and the injector would know how to create an instance of it when we need to inject what we are going to do next. We are going to add `StudentSvc` in the list of `viewProviders` of the root componen

```
@Component({
  selector: 'app',
  templateUrl : 'templates/app.tpl.html',
  viewProviders: [StudentSvc] // <- registering the service
})
```

The last thing we need to do is to inject the service in the constructor of our root component:

```
1  class Root  {
2    private name: string;
3    private students: any[];
4    constructor (studentSvc: StudentSvc) { // <- injecting the service
5      this.name = 'Simple Service Demo';
6      this.students = studentSvc.getAll(); // <- calling the `getAll` method
7    }
8  }
```

- In the constructor, we are defining a variable to be of type `StudentSvc`. By doing that the injector will create an ins `StudentSvc` to be used
- And on line 6 we are calling the `getAll` method from the service to get a list of all students.

Finally, we can verify that the `getAll` method is actually called by printing the students in the template:

**`app.tpl.html`**

```
<h1>{{ name }}</h1>

<ul>
  <li *ngFor="#student of students">Name: {{ student.name }}, id: {{ student.id }}</li>
```

```
</ul>
```

and it would output:

```
Name: Tom, id: 1
Name: John, id: 2
Name: Kim, id: 3
Name: Liz, id: 4
```

## Data and State Management

- Angular is flexible and doesn't prescribe a recipe for managing data in your apps
- Since observables are integrated into Angular, you can take advantage of observables to manage data and state
- You ca use services to manage streams that emit models
- Components can subscribe to the streams maintained by services and render accordingly.
    - For example, you can have a service for a Todo app that contains a stream of todos and a `ListComponent` can listen for t when a new task is added.
    - You may have another component that listens for the user that has been assigned to a task provided by a service.
- The steps for creating different parts of an app can be summarized in three steps:
    - Defining a Model using a class
    - Defining the service
    - Defining the component

## Observables

- Observables can help manage async data
- Observables are similar to Promises but with a lot of differences
- Observables emit multiple values over time as opposed to one
- Angular embraces observables using the RxJS library.
- Observables emit events and observers observe observables.
- An observer *subscribes* to events emitted from an observable.
- RxJS has an object called *subject* that can be used both as an observer or an observable. *Subject* can be importec very easily:

```
import {Subject} from 'rxjs/Subject';
```

- A subscription can be canceled by calling the `unsubscribe` method.

**TODO**

## State Management with Observables

- There are several ways to manage state, one of them is using observables
- Observables can be used to represent the state of the app
- Changes in the state are represented as an observable

**TODO**

## Http

- Using the `Http` class, you can interact with API endpoints

- Http is available as an injectable class
- `Http` has a request method that returns an Observable which will emit a single Response when a response is rece
- You can inject `http` in the constructor of a class: `constructor(http: Http) {...}`

## Getting Data from Server

In this section we are going to use the `http` class to get a list of students from a server by hitting `/api/students`

### Project Files

The project files for this section are in angular2-intro/project-files/angular-examples/http/get-students

### Getting Started

Before anything, let's add the `http.js` file from Angular's bundle. In your `index.html` file add the following to the h

```html
<script src="/node_modules/angular2/bundles/http.js"></script>
```

After that, we are going to make a service that handles getting data from the endpoint. We are going to call this `Stuc`

```
1 @Injectable()
2 class StudentSvc {
3   constructor(private http: Http) {} /* Inject Http */
4   getStudents(): Observable<Response> {
5     return this.http.get('/api/students');
6   }
7 }
```

- On line 1, we are using the `Injectable` decorator to make our class injectable
- In the constructor we are injecting the `Http` service and making a reference to it in a private variable `http`
- The `getStudents` method makes a `GET` call to our local endpoint an returns an `Observable`

Now that we have the `StudentSvc` service, we can create a component and inject the `StudentSvc` to it:

```
@Component({
  selector: 'app',
  templateUrl :'templates/app.tpl.html',
  providers: [StudentSvc] // <- adding to the list of providers
})
```

In addition to the `StudentSvc`, we also need to add `HTTP_PROVIDERS` in the providers array:

```
@Component({
  selector: 'app',
  templateUrl :'templates/app.tpl.html',
  providers: [HTTP_PROVIDERS, StudentSvc] // <- adding `HTTP_PROVIDERS`
})
```

After adding the providers, we can define the component class:

```
@Component({...})
class HttpGetExample  {
  private name: string;
  private students: Observable<Response>;
  constructor (studentSvc: StudentSvc) {
    this.name = 'HTTP Get';
    studentSvc.getStudents().subscribe(resp => this.students = resp.json());
  }
}
```

If you notice, we are injecting the `StudentSvc` in the constructor and we are calling the `getStudents` method in the
The `getStudents` returns an observable that we can subscribe to get the data out as they arrive. We also call the `j`
each response to get the JSON data.

After getting the data, we can print the result in the view:

`app.tpl.html`

```
<h1>{{ name }}</h1>
<ul>
  <li *ngFor="#student of students">
    {{ student.name }}, {{ student.lastname }}
  </li>
</ul>
```

Here we are using the built-in `ngFor` directive to loop through the array of students and print their name and last nar

## Working with Forms

Angular has convenient methods for working with forms, including validation.

**TODO**

## Angular Router

Angular has a stand-alone module responsible for handling routing.

**TODO**

## Unit Testing

Unit testing with Angular requires some set up. First, let's gather all the libraries and modules that we need.

**TODO**

# Deep Dive

Let's deep dive into Angular and RxJS concepts

## Components in Depth

- A component declares a reusable building block of an app
- A TypeScript class is used to define a component coupled with the `@component` decorator

The `@component` decorator defines the following:

- selector: `string` value defining the css selector targeting an html element
- inputs: `array of string` values defining the inputs to the component
- outputs: `array of string` values defining the output of the component
- properties: `array of string` values defining the properties

- events: `array of string` values defining the events
- host?: {['string']: 'string'},
- providers: `array of objects` : defines the set of injectable objects that are visible to a directive/component and its children.
- exportAs: `string` value defining the exported value
- moduleId: `string` value defining the module id
- viewProviders?: `array of objects` : defines the set of injectable objects that are visible to a directive/component v children.
- queries: {[key: string]: any},
- changeDetection: `ChangeDetectionStrategy` object defining the strategy for detecting changes:
    - `ChangeDetectionStrategy.Default` : sets detector mode to `CheckAlways`
    - `ChangeDetectionStrategy.OnPush` : sets detector mode to `CheckOnce`
    - `ChangeDetectionStrategy.Detached` : change detector sub tree is not a part of the main tree and should be skipped
    - `ChangeDetectionStrategy.CheckAlways` : after calling detectChanges the mode of the change detector will remain `Check`
    - `ChangeDetectionStrategy.Checked` : change detector should be skipped until its mode changes to `CheckOnce`
    - `ChangeDetectionStrategy.CheckOnce` : after calling detectChanges the mode of the change detector will become `Checke`
- templateUrl: `string` value for the url path to the template
- template: `string` value for the template
- styleUrls: `array of string` values defining url paths to css files
- styles: `array of string` values defining css styles:
    - styles: ['.myclass { color: #000;}'],
- directives: `array` of directives used in the component
- pipes: `array` of pipes used in the component
- encapsulation: `ViewEncapsulation` value that defines template and style encapsulation options:
    - `ViewEncapsulation.None` : means do not provide any style encapsulation
    - `ViewEncapsulation.Emulated` : No Shadow DOM but style encapsulation emulation using extra attributes on the DOM (de
    - `ViewEncapsulation.Native` : means provide native shadow DOM encapsulation and styles appear in component's templa shadow root.

**TODO**

# RxJS

- RxJS is a library for reactive programming
- Reactive programming is a natural way of thinking about asynchronous code which concerns itself with operations streams
- RxJS is used in this paradigm to compose asynchronous data/event streams
- Using the reactive paradigm, you can perform operations on streams using a consistent interface regardless of the
- Streams of events/data are known as Observables, i.e. a data/event stream = observable
- A program can be just composed of different data streams (observables)

**TODO**

- Visualization