Notes

- The book assumes that you are working in a Unix-like environment. If you are on Windows you can use 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 ca check out the project files. Throughout the book, you will see references to the project files. Those refer t example, `angular2-intro/project-files/hello-angular` refers to the `hello-angular` folder inside th

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

- 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 Nod correctly.

- All the keyboard shortcuts are mac-based. But if you are using a non-mac machine, you can almost alw `ctrl` and you should be good. For example, if you a see a shortcut like `command + shift + b`, you ca 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 sh

```
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 vers

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

```
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 erro `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 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 c

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
- 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 th 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 `angular2-intro/project-files/vscode`. You can either follow along or check out the folder to see the

Installing TypeScript

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

```
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 lik

```
message TS6029: Version 1.7.5
```

In addition to the compiler, we also need to install the TypeScript Definition manager for DefinitelyTyped (

```
npm i tsd -g
```

Using TSD, you can search and install TypeScript definition files directly from the community driven Defini
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` :

```
1 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 desk
- 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

```
1 {
2   "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 `.vsc` 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`

```
> run build tasks
```

you can also use the `command + shift + b` keyboard shortcut instead. This will start the debugger and 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 brea
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 i

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

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

Debugging App from VSCode

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

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 cr
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 executabl debugger you need to have a server running serving the app. You can change the `url` value according `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 an everything is set up correctly, you should see an instance of chrome running in incognito mode. To trigge 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 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 hav Angular code that you will write. Knowing TypeScript definitely helps to understand Angular, but again it 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 any valid JavaScript code (ES 2015/2016...) is valid TypeScript code. You can basically change the exte 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
- Interfaces also define custom types

Basic Interface

Below is an example of an Interface that defines two properties and three methods that implementers sh

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 implementatio
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 ca
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 `Po`
`Point` by adding a third field. An then we create a point of type `point3d` and assign a value to it. We r
`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
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 se
`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 c

```
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

```
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 t

- 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 obje
`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 class.
- `public` is the default modifier which means the property or the method is the accessible everywhere a anyone.

Adding a constructor

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

Let's add a constructor to the `Car` class that initializes the `distance` value to 0. This means that all the 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 v 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:

```
1 class Car {
2   constructor (private distance) {
3   }
4 }
```

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

```
1 var Car = (function () {
2   function Car(distance) {
3     this.distance = distance;
4   }
5   return Car;
6 })();
```

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

```
1 class Car {
2   constructor (private distance = 0) {
3   }
4   getDistance():number { return this.distance; }
5 }
```

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

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

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

```
1 const mycar = new Car(5);
2 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 ve that. Let's take our example above and add a setter and getter for the distance property. But before that `distance` to `_distance` to make it explicit that it is private. It is not required but it is a common patterr with an underscore.

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

```
4 }
```

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

```
1 class Car {
2   constructor (private _distance = 0) {}
3   get distance() { return this._distance; }
4 }
```

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, tha
Below is the relevant generated JavaScript:

```
1 Object.defineProperty(Car.prototype, "distance", {
2   get: function () { return this._distance; },
3   enumerable: true,
4   configurable: true
5 });
```

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

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

```
1 class Car {
2   constructor (private _distance = 0) {}
3   get distance() { return this._distance; }
4   set distance(newDistance: number) { this._distance = newDistance; }
5 }
```

Now we can both get and set the distance value:

```
1 const coolCar = new Car();
2 console.log(coolCar.distance); // -> 0
3
4 coolCar.distance = 55;
5 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.isArra`
through the `Array` but not an instance of an array:

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

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

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

```
1  class Car {
2    static controls: {isAuto: boolean } = {
3      isAuto: true
4    };
5    static isAuto():boolean {
6      return Car.controls.isAuto;
7    }
8    constructor (private _distance = 0) {}
9    get distance() { return this._distance; }
10 }
11
12 console.log(Car.controls); // -> { isAuto: true }
13 console.log(Car.isAuto()); // -> true
```

- On line 2 we are defining a static property called `controls` using the `static` modifier. Then we spec a value for it.
- On line 5 we are defining a static method called `isAuto` using the the `static` modifier. This method `isAuto` from the static `control` object. Not that we get access to the class using the name of the clas `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

```
1  interface ICarProps {
2    distance: number;
3  }
4  interface ICarMethods {
5    move():void;
6  }
```

Making the `Car` class implement the interfaces:

```
1  class Car implements ICarProps, ICarMethods {
2    distance: number;
3    constructor () {
4      this.distance = 5;
5    };
6    move():void {
7      this.distance += 1;
8    };
9  }
```

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

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

Inheritance

In Object-oriented programming, a class can inherit from another class which helps to define shared attril objects. Although this pattern is very useful, it should be used cautiously as it can lead to code that is har more about classical inheritance and prototypical inheritance by watching Eric Elliot's talk at O'Reilly's Flu 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 ot 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, c the setters and getters for them. Additionally, we are defining the `toString` method that JavaScript inte 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` 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` keyword.
- They both call the `Vehicle` 's constructor in their own constructor method before implementing their c `constructor(name?: string) { super(); }`
- They both take an optional `name` property to set the name of the vehicle. If not name is provided, it wi '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`

TODO

- `constructor overloading`

Class Decorators

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

TODO

`add content`

Modules

- In TypeScript you can use modules to organize your code, avoid polluting the global space, and expos use.
- Multiple modules can be defined in the same file. However, it makes more sense to keep on module pe
- 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" />` 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` , `MyModul`

- 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 Typ 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 whe
- 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 module
  - 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 i from the vehicle class. Then we are going to expose the car class to the outside world and import it from 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 a

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 us

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 u
file and import it into the main file. Let's create a file called `MyModule.ts` and move the module definitio
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
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. E
make some changes to our config files. First, we are going to add the `out` property in the `tsconfig.js`
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 a
`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 dropdown and run the debugger and it should stop if you put a breakpoint in any of your Type

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

Splitting Internal Modules

Internal modules in TypeScript are open ended. This means that you can define a module with the same keep adding to it. This is also known as merging. In this section we are going to demonstrate merging mu 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` m

```
// 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 adc 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 v
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 w
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 t
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 followin

```
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 `so` 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 p 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, pipes, services, events and other concepts. By the end of the chapter you should have a basic understar 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 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:st32lth/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-exam` 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 y `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 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` **(nee**
- 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 r 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

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

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

TODO

Common Interfaces

- `OnInit`

TODO

Common Enums

- `ChangeDetectionStrategy`

TODO

Metadata Classes

- 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 system

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

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 closes
- 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 conte
- ViewQuery: Similar to `QueryMetadata`, but querying the component view, instead of the content childr

TODO

Component Basics

Components are at the heart of Angular. The main idea is that you break down your application into differ and let the components handle the rest. Every component has a controller defined by a class and a temp addition, a component's job is to enable the user experience and delegate everything non-trivial to servic

In this section we are going to write a simple `HelloAngular` component, compile it and run it in the brov 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 just keep things simple.

The project files for this chapter are in `angular2-intro/project-files/angular-examples/basic-compon` 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 <st32lth@gmail.com> (http://github.com/st32lth)",
  "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>
<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:

```
<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 ma `HelloAngular` and then bootstrap the app with it:

`main.ts`

```
import {Component, OnInit} from 'angular2/core';
```

```
import {bootstrap} from 'angular2/platform/browser';

@Component({
  selector: 'app',
  styles: [`h1 { line-height: 100vh; text-align: center }`],
  template: `<h1>{{ name }}</h1>`
})
class HelloAngular implements OnInit {
  name: string;
  constructor() { this.name = 'Hello Angular'; }
  ngOnInit() { console.log('component linked'); }
}

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 technica 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 cc `<app></app>` tag, it is going to load the template (on line 6) and instantiates the class for it (defined on li
- On line 9, we are defining a class called `HelloAngular` that defines the logic of the component. And f the `OnInit` interface to log something to the console when the component is ready with its data. We wi lifeCycle hooks later.
- Last but not least, we call the `bootstrap` method with the `HelloAngular` class as the first argument 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. 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 +` 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 r
- 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 `in` `<!-- 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>
```

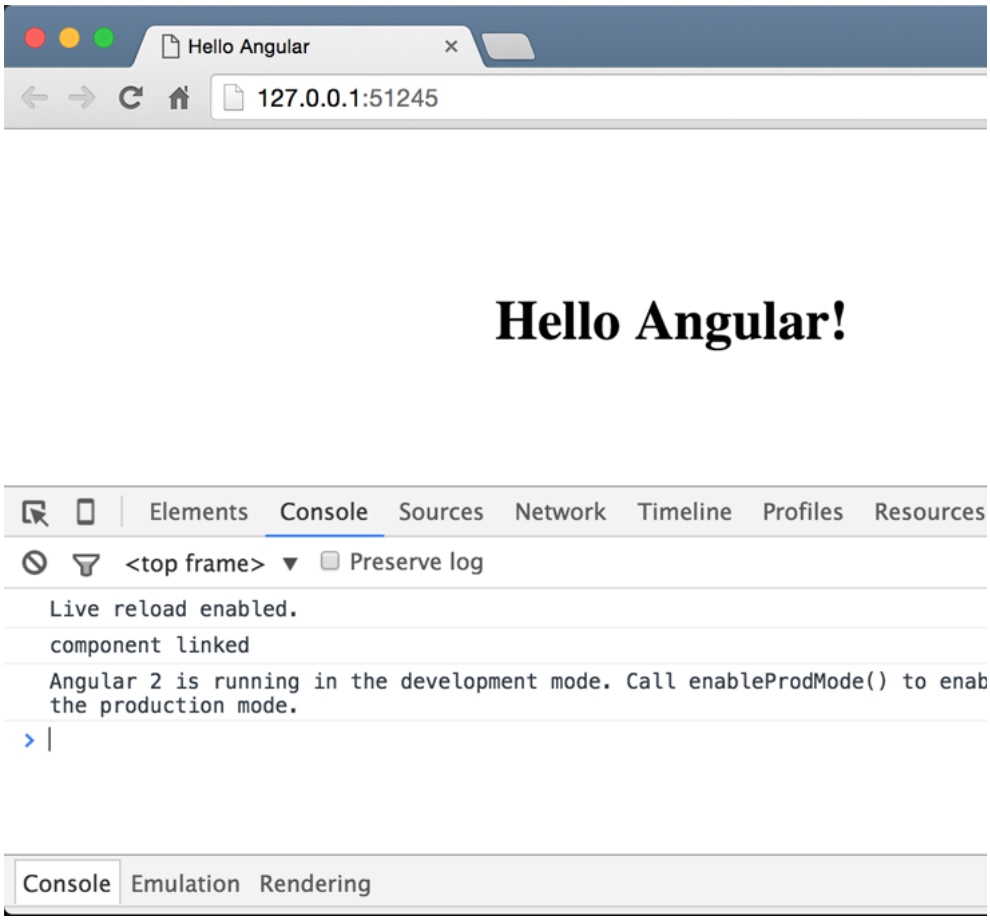
Now we can use our component in the body of the html:

```
<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:



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 App from VSCode section in case you missed to install it. But, assuming that you have the extension inst app from VSCode. In order to do that, we need to create a `launch.json` file in the `.vscode` folder:

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

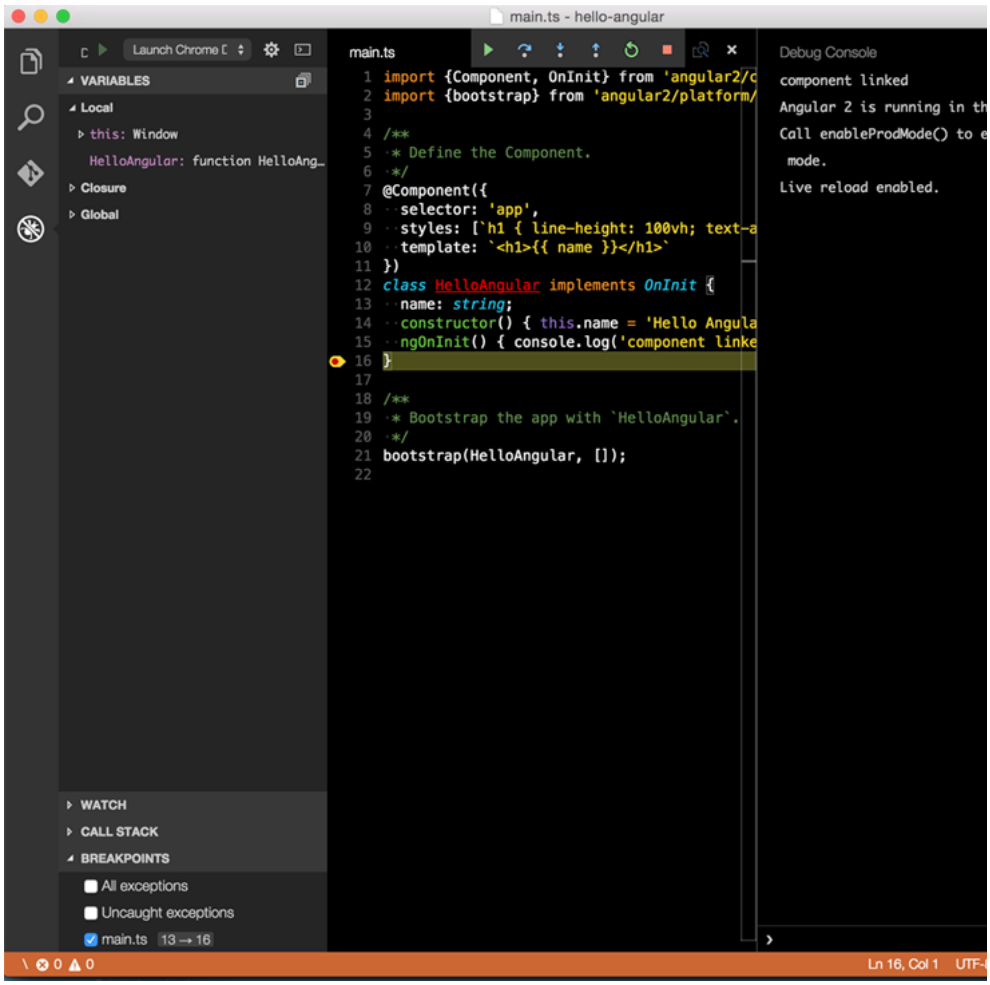
After you created the file, put in the following configuration in the file:

```
{
  "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 Chrom
      "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
- Make sure that the `runtimeExecutable` path is valid. This value would be different depending on your
- Make sure that the `url` value is valid as well. The `url` value has to match the path that you see whe 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 icon or hit F5 on the keyboard. After that, an instance of Chrome should be opened in incognito mode. In debugger just refresh the page and you should be able to see the debugger pausing in VSCode. If everyt should be able to see something like the following screenshot:



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`
- 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`

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:

```
@Component({
  selector: 'user',
  template: '<h1>Hello, {{ name }} {{ lastName }} id: {{ userId }}</h1>',
  inputs: ['name', 'lastName', 'userId'] // <- specifying the inputs to the `User` comp
})
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 `U`
`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 instanc
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 t component and create a new component called `Permission`. Then we are going to use the the `Permis` `User` component and set the `uid` of `Permission` by the `userId` of the `User`.

The `Permission` component is defined as follows:

```
@Component({
  selector: 'permission',
  template: '<h2> Restriction is: {{ restriction }}'
})
class Permission {
  @Input() private uid: string;
  private restriction: string;
  constructor() {
    this.restriction = 'none';
  }
  ngOnInit() {
    this.restriction = this.uid === '1' ? 'admin' : 'normal';
  }
}
```

- 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 provi 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 i

then we are going to register the `Permission` component with the `User` component so that we can us

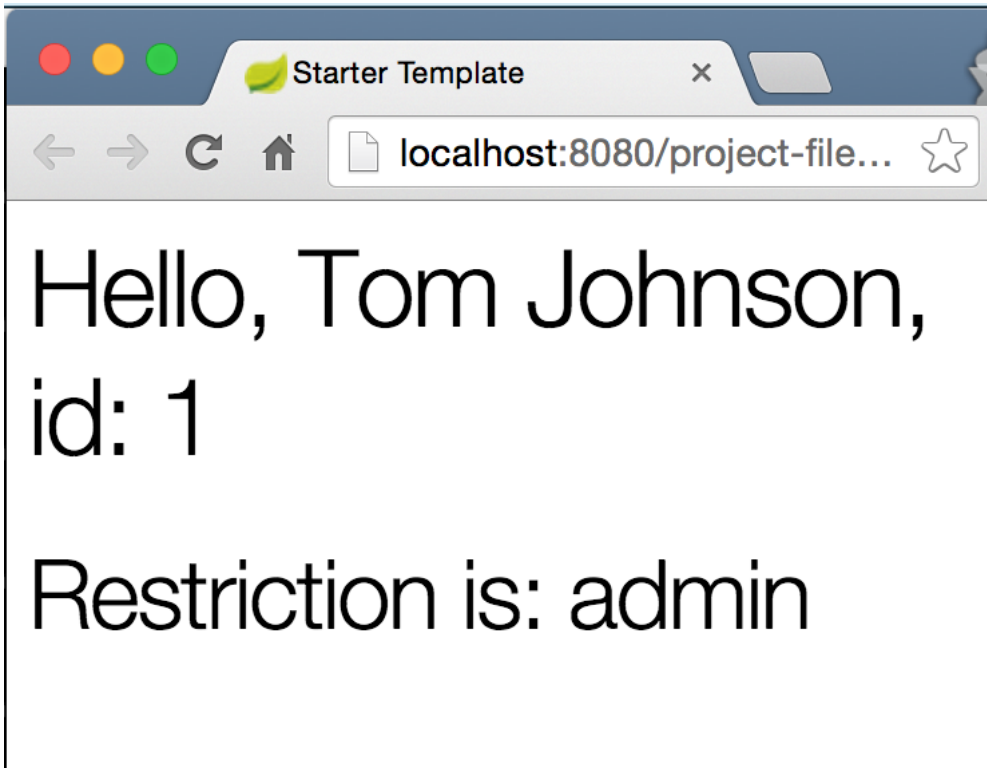
```
@Component({
  selector: 'user',
  ///...
  directives: [Permission] // <-
})
class User {}
```

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` com

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



Input to components

Binding to DOM Properties

- `[style.color]="done ? 'green' : 'red' "`
- `[class.name]="done ? 'done' : 'pending'"`
- `[hidden]=isLoaded ? true : false`

TODO

Component Output/Events

- Events can be emitted from components. These events can be either custom or they could be DOM ev
- The syntax is `(eventname)="fn()"` where `eventname` is the name of the event and `fn` is the handle
- 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 `
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-e

Final Result

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

```
1 <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 add able to hook into a custom event and run the `addOne` method whenever the event is fired:

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

Getting Started

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

```
@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 th

Using EventEmitter

Now, let's see how we can use the `EventEmitter` to increment the value by one every time a custom ev order to achieve that, we are going to create an attribute directive called `AdderAuto`. Start by importing 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` attr
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
`(click)`, we want to be able to use `(myevent)`. To achieve that, we need to create an instance variab
`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 em
- 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 secon

```
@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 whene
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.

Native DOM Output Events

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 template
- 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 sel
- 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 and Shadow DOM Basics

TODO (shadow dom, light dom,