MedTech – Mediterranean Institute of Technology

CS-Web and Mobile Development

Chp5- Angular

ES6 and TypeScript, Components, Dependency Injection…

Why Angular?

Angular

- Angular JS
  - Javascript Framework for creating web and mobile single page applications
- Angular 2
  - Easier to learn than Angular 1.x
  - Fewer concepts
  - Supports multiple languages (ES5, ES6, TypeScript and DART)
  - Modular (everything is a Component)
  - Performant (5X faster than version 1)
- This document explains the notions of Angular2, RC6 (February 2017)

Angular

TYPESCRIPT

Description

TypeScript

- Angular 2 is built in TypeScript
- Official collaboration between Microsoft and Google
- JavaScript-like language
  - Superset of EcmaScript6
- Improvements over ES6
  - Types
  - Classes
  - Annotations
  - Imports
  - Language Utilities (e.g. destructuring)

Types

TypeScript

- Major improvement over ES6: type checking
  - Helps when writing code because it prevents bugs at compile time
  - Helps when reading code because it clarifies your intentions
- Typing is optional
- Same types as in ES: string, number, boolean,…

```
var name: string;
```

- Types can also be used in function declarations:

```
function greetText(name: string): string{
    return "Hello " + name;
}
```

Built-in Types

TypeScript

Types	Examples
String	`var name : string = 'Lilia'`
Number	`var age : number  = 36`
Boolean	`var married : boolean  = true`

Array	`var jobs : Array<string> = [‘IBM’, ‘Microsoft’, ‘Google’]`
	`var jobs : string[]  = [‘Apple’, ‘Dell’, ‘HP’]`

Enums	`enum Role {Employee, Manager, Admin};`
	`var role: Role = Role.Employee;`
	`Role[0]       //returns Employee`

Any
(default type if omitting typing for a given variable)
	`var something: any = 'as string';`
	`something = 1;`
	`something = [1, 2, 3];`

Void
(no type expected, no return value)
	`function setName(name: string): void {`
	`    this.name = name;`
	`}`

Classes

TypeScript

- In ES5, OO programming was accomplished by using prototype-based objects
- In ES6, built-in classes were defined

```
class Vehicle {}
```

- Classes may have properties, methods and constructors
- Properties
  - Each property can optionally have a type

```
class Person {
    first_name: string;
    last_name: string;
    age: number;
}
```

Classes

TypeScript

- Methods
  - To call a method of a class, we have to create an instance of this class, with the new keyword

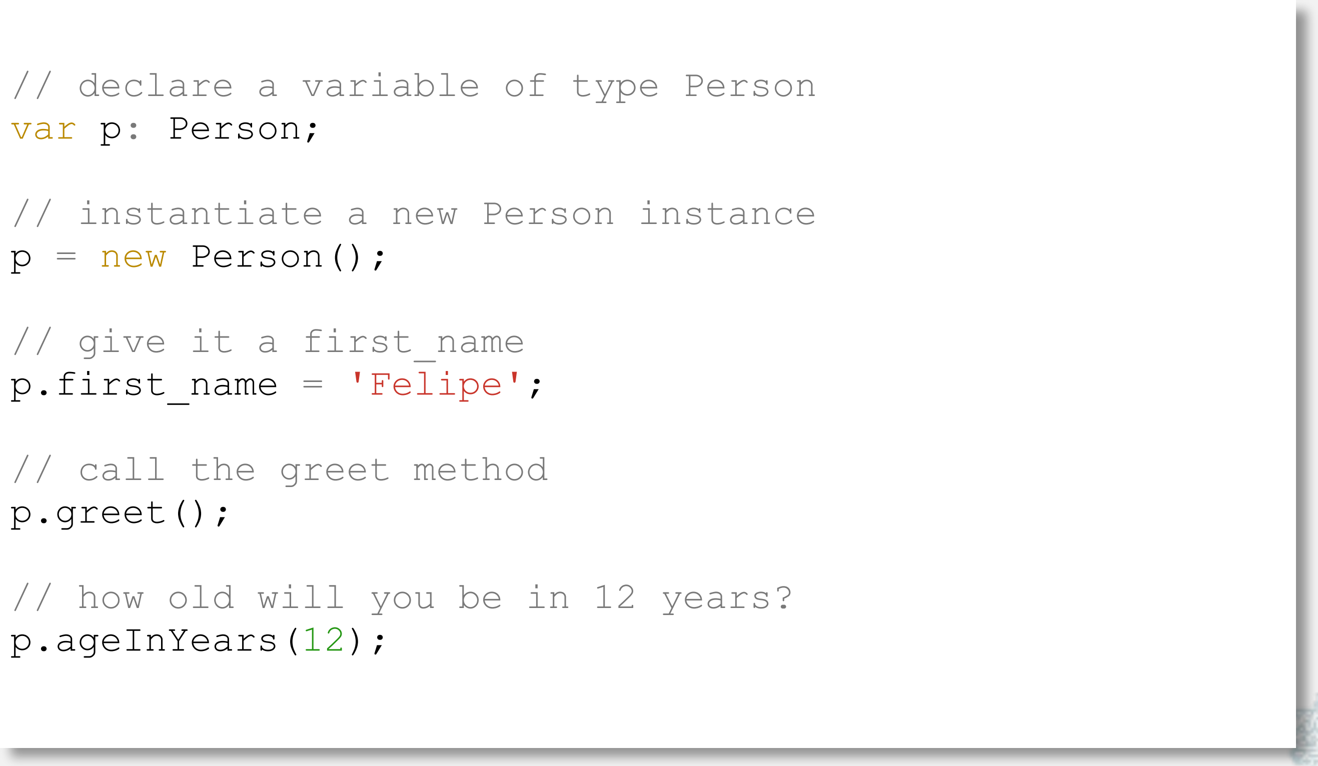
```
class Person {
    first_name: string;
    last_name: string;
    age: number;

    greet(){
        console.log("Hello ", this.first_name);
    }
    ageInYears(years: number): number {
        return this.age + years;
    }
}
```

  - If the methods don't declare an explicit return type and return a value, it's assumed to be any

Classes

TypeScript

- Methods
  - To invoke a method:

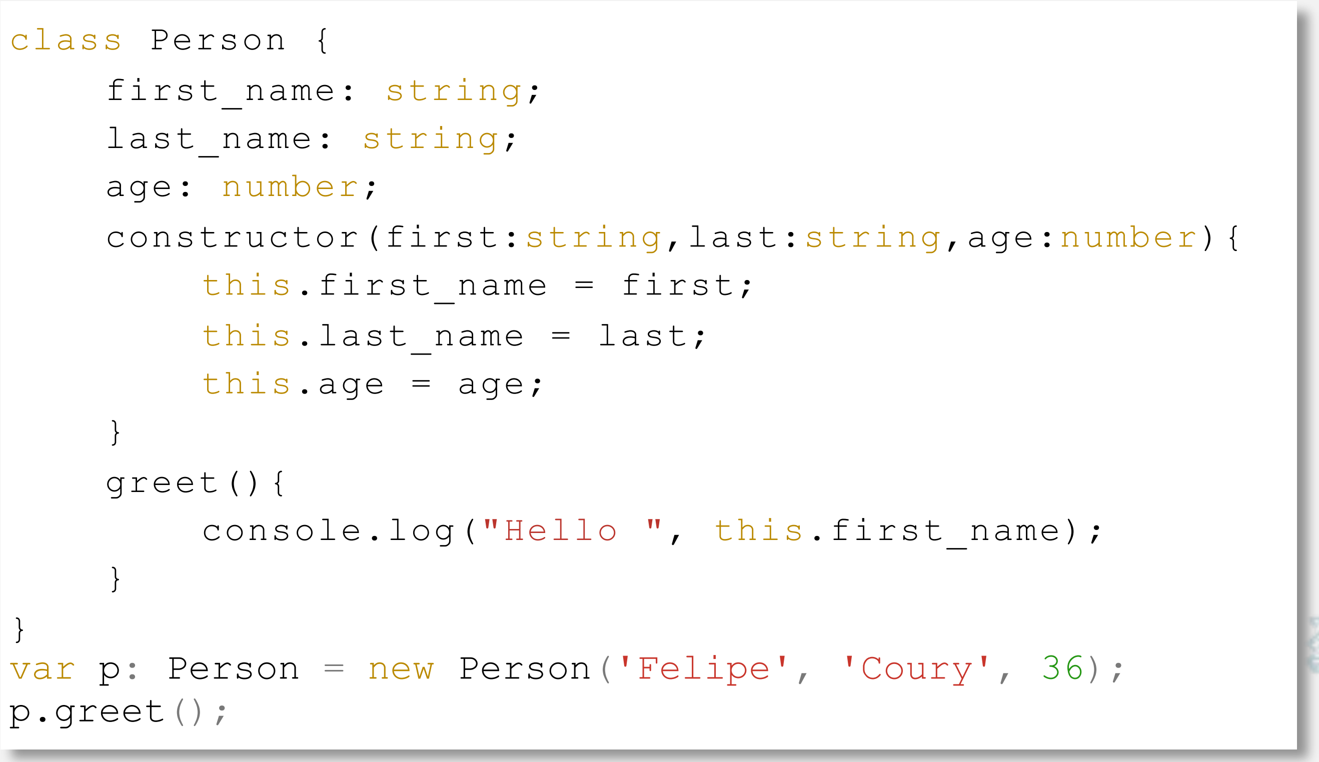


```
// declare a variable of type Person
var p: Person;

// instantiate a new Person instance
p = new Person();

// give it a first_name
p.first_name = 'Felipe';

// call the greet method
p.greet();

// how old will you be in 12 years?
p.ageInYears(12);
```

Classes

TypeScript

- Constructor
  - Named constructor(..)
  - Doesn't return any values

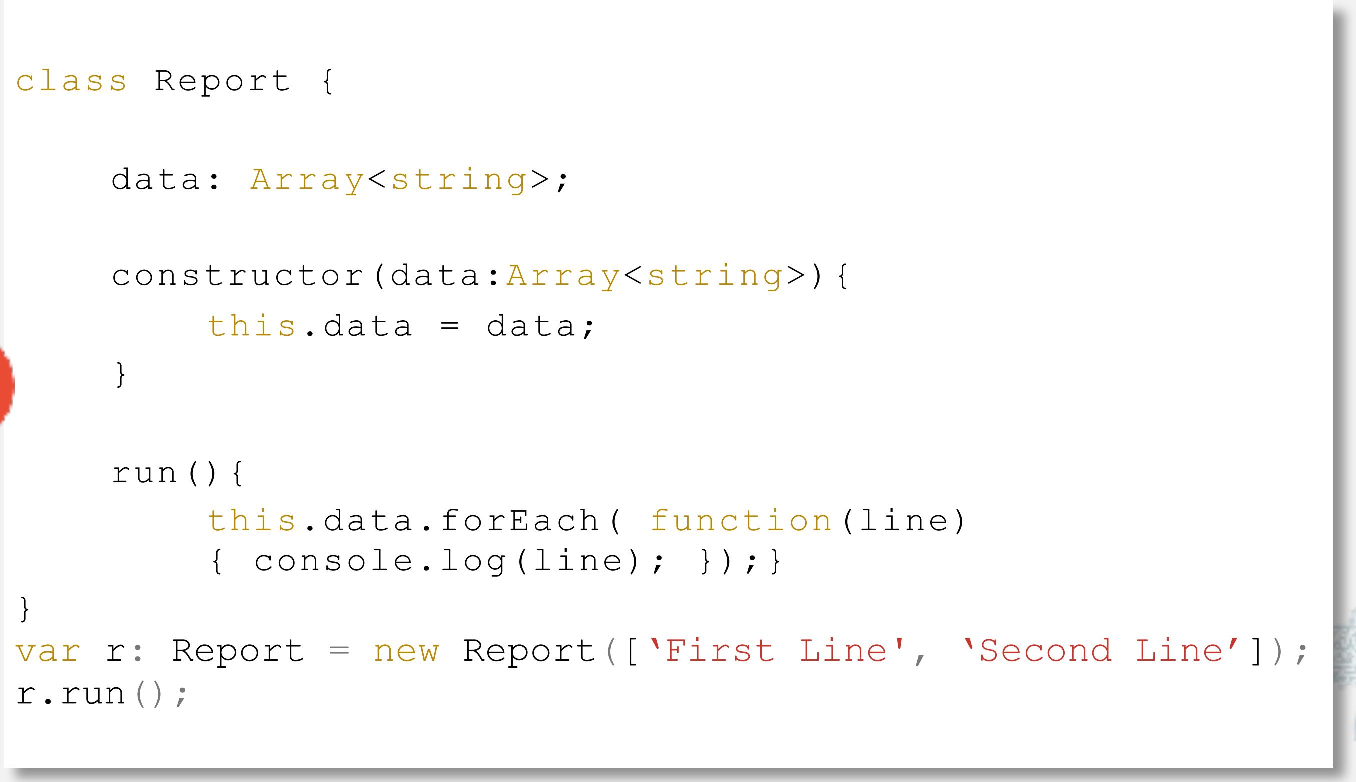


```
class Person {
     first_name: string;
     last_name: string;
     age: number;
     constructor(first:string,last:string,age:number){
          this.first_name = first;
          this.last_name = last;
          this.age = age;
     }
     greet(){
          console.log("Hello ", this.first_name);
     }
}
var p: Person = new Person('Felipe', 'Coury', 36);
p.greet();
```

Inheritance

TypeScript

- Inheritance is built in the core language
- Uses the extends keyword
- Let’s take a Report class:

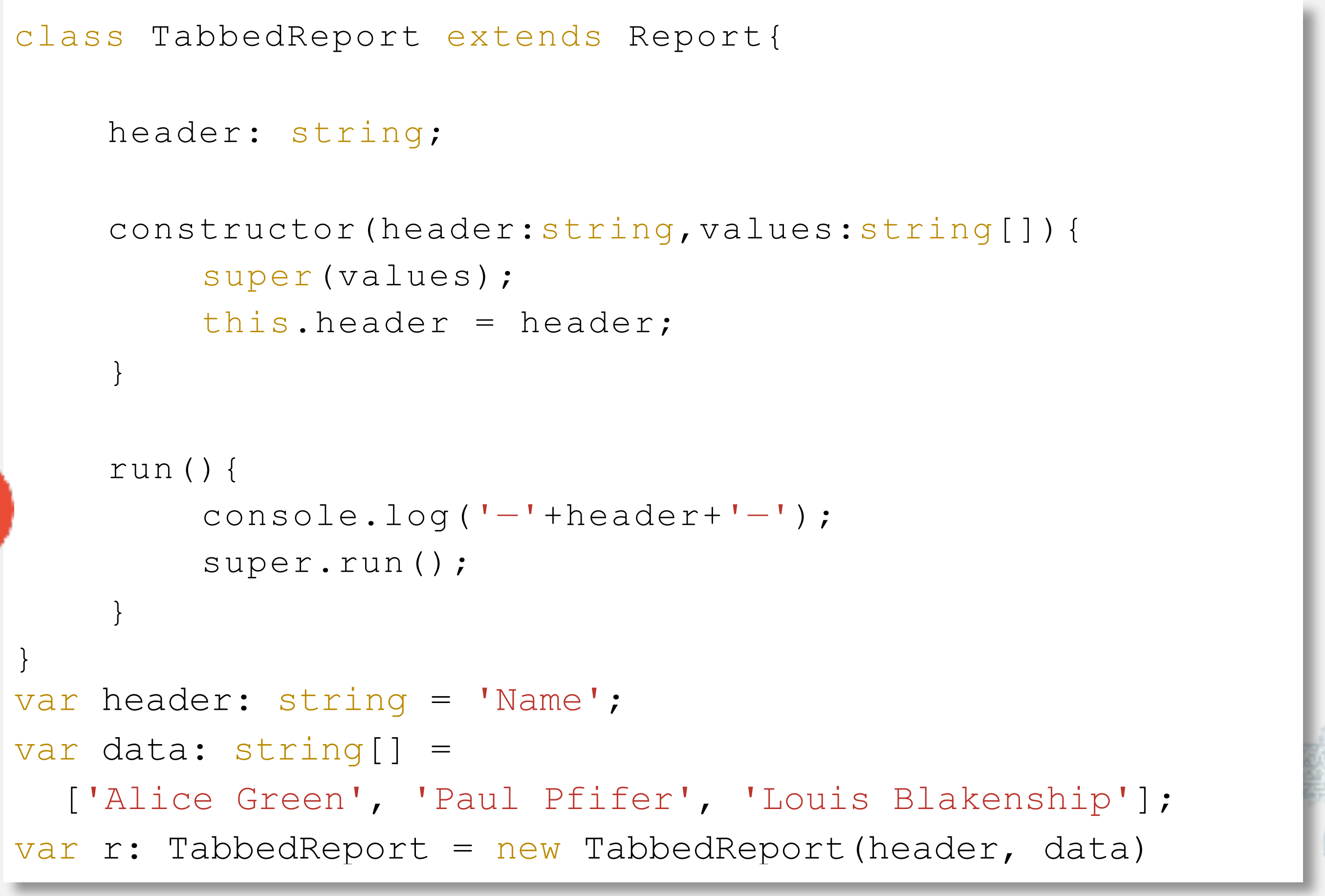


```
class Report {

    data: Array<string>;

    constructor(data:Array<string>){
        this.data = data;
    }

    run(){
        this.data.forEach( function(line)
        { console.log(line); });}
}
var r: Report = new Report([‘First Line', ‘Second Line’]);
r.run();
```

Inheritance

TypeScript

- We want to change how the report presents the data to the user:

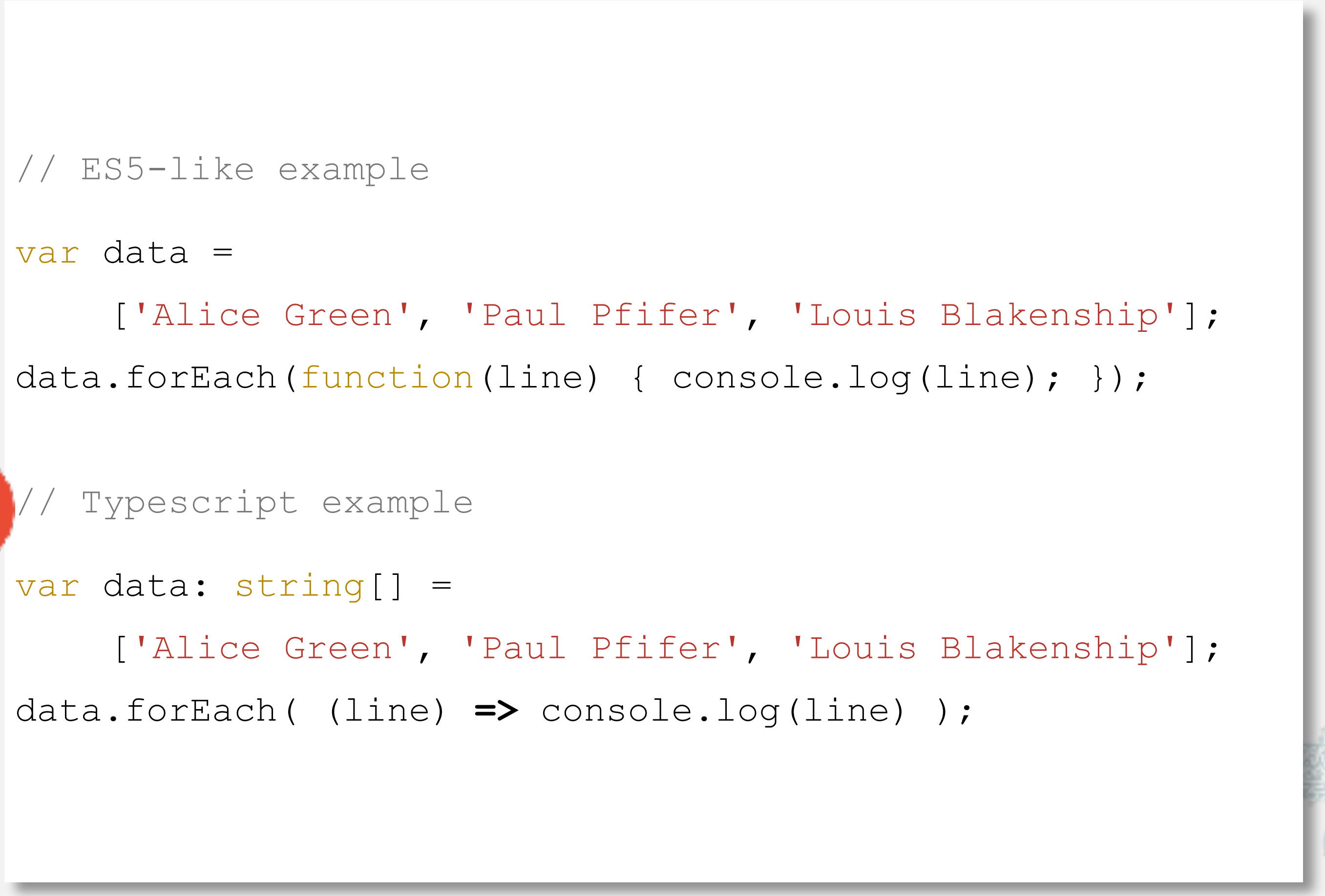


```
class TabbedReport extends Report{

    header: string;

    constructor(header:string,values:string[]){
        super(values);
        this.header = header;
    }

    run(){
        console.log('-'+header+'-');
        super.run();
    }
}
var header: string = 'Name';
var data: string[] =
  ['Alice Green', 'Paul Pfifer', 'Louis Blakenship'];
var r: TabbedReport = new TabbedReport(header, data)
```

Fat Arrow Functions

TypeScript

- Fat arrow => functions are a shorthand notation for writing functions

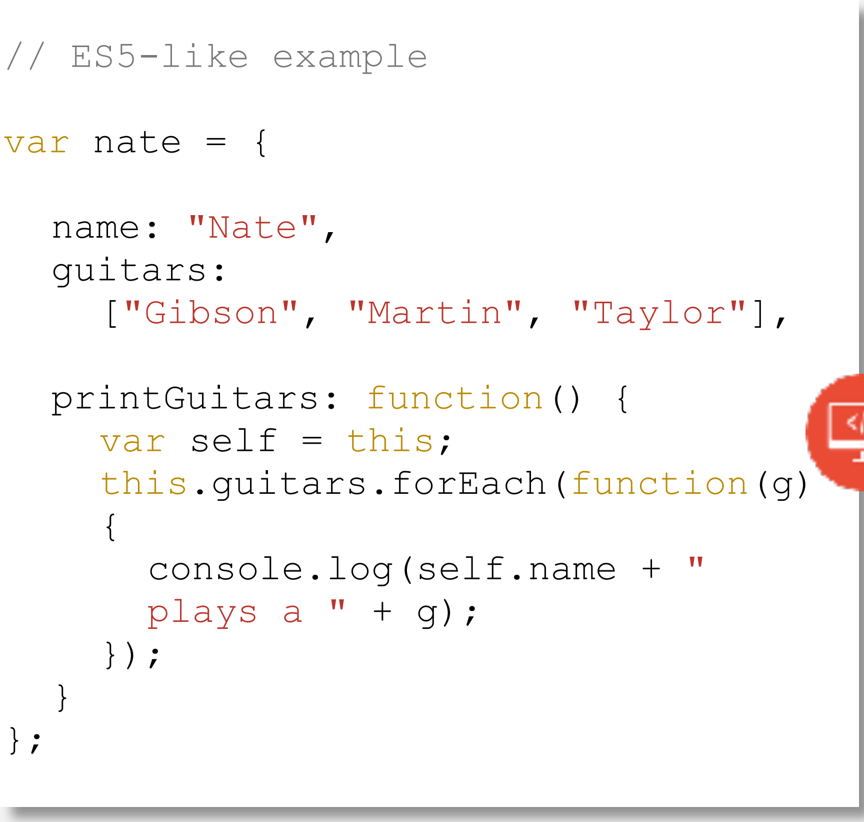


```
// ES5-like example
var data =
     ['Alice Green', 'Paul Pfifer', 'Louis Blakenship'];
data.forEach(function(line) { console.log(line); });

// Typescript example
var data: string[] =
     ['Alice Green', 'Paul Pfifer', 'Louis Blakenship'];
data.forEach( (line) => console.log(line) );
```

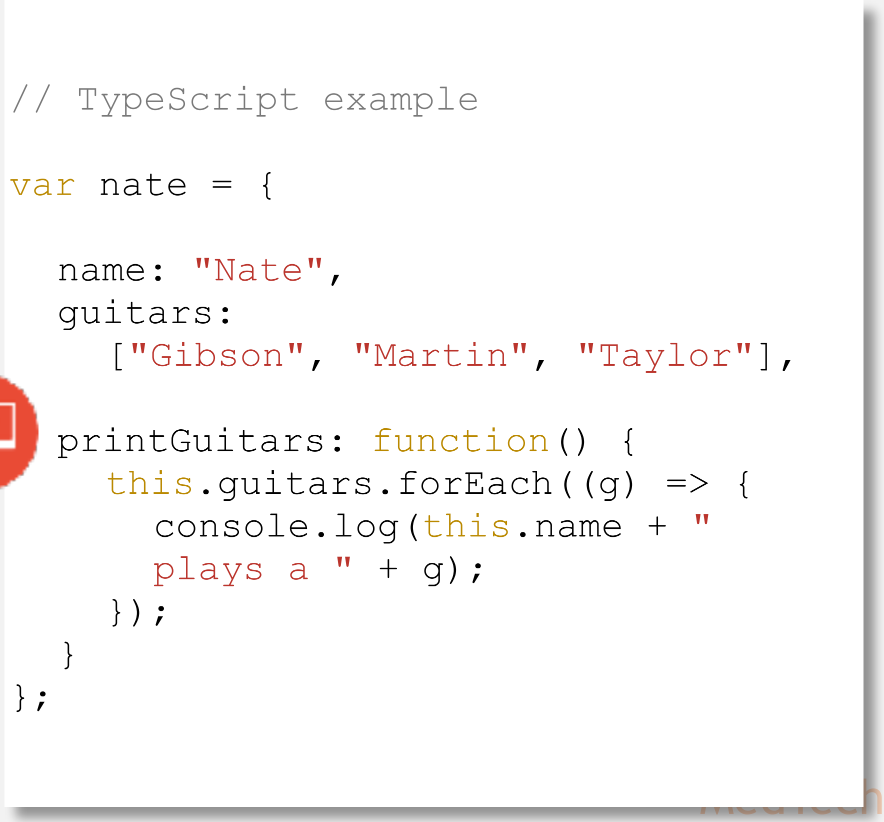
Fat Arrow Functions

TypeScript

- The => syntax shares the same this as the surrounding code
  - Contrary to a normally created function in JavaScript



```
// ES5-like example

var nate = {

  name: "Nate",
  guitars:
    ["Gibson", "Martin", "Taylor"],

  printGuitars: function() {
    var self = this;
    this.guitars.forEach(function(g)
    {
      console.log(self.name + "
      plays a " + g);
    });
  }
};
```

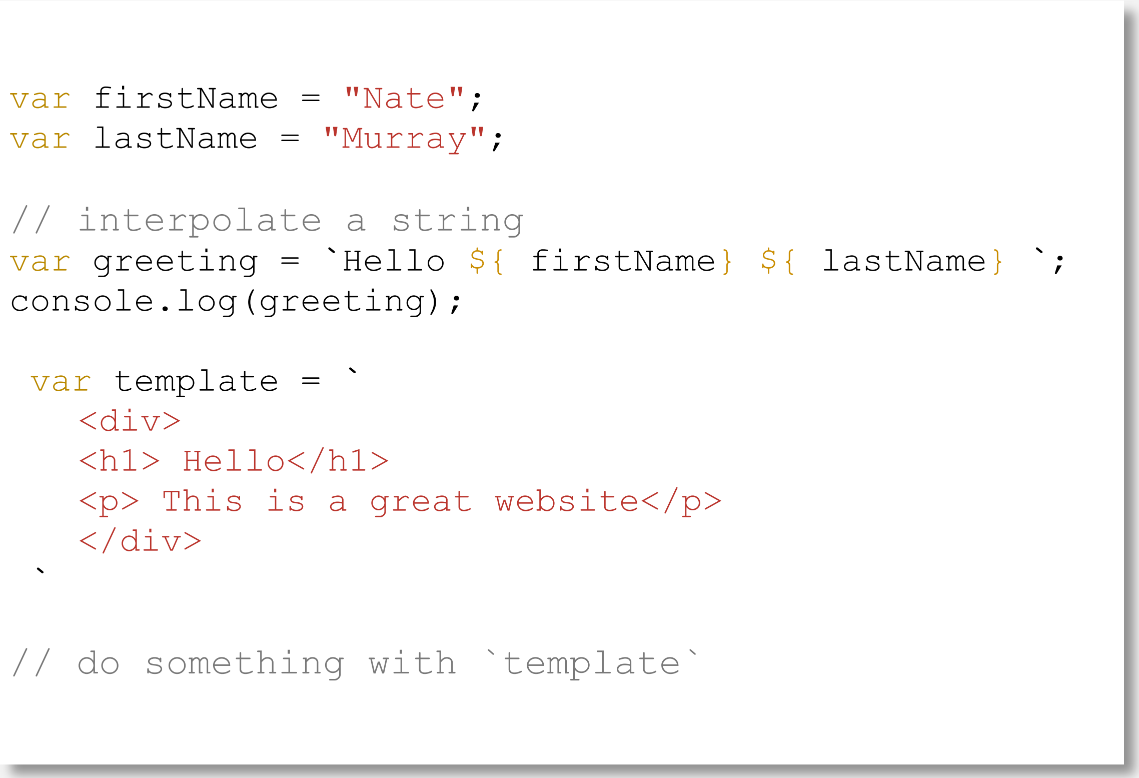


```
// TypeScript example

var nate = {

  name: "Nate",
  guitars:
    ["Gibson", "Martin", "Taylor"],

  printGuitars: function() {
    this.guitars.forEach((g) => {
      console.log(this.name + "
      plays a " + g);
    });
  }
};
```

Template Strings

TypeScript

- Introduced in ES6, enable:
  - Variables within strings, without concatenation with +
  - Multi-line strings



```
var firstName = "Nate";
var lastName = "Murray";

// interpolate a string
var greeting = `Hello ${ firstName} ${ lastName} `;
console.log(greeting);

 var template = `
   <div>
   <h1> Hello</h1>
   <p> This is a great website</p>
   </div>
 `

// do something with `template`
```

TypeScript Language

TypeScript

- And there is more…
- Consult: http://www.typescriptlang.org/docs/tutorial.html for more detailed information about the language!

Angular

COMPONENTS IN ANGULAR

Angular Application Structure

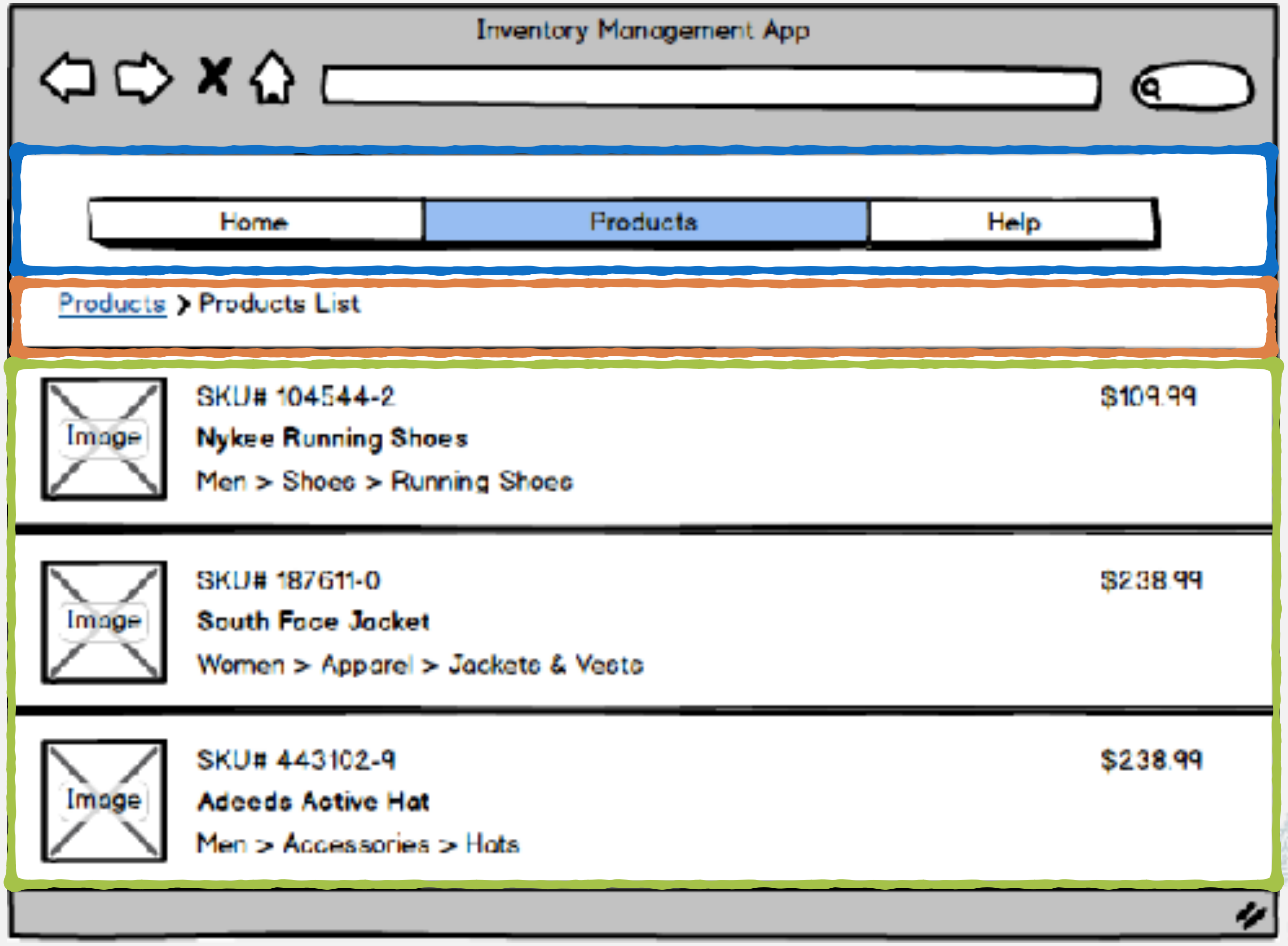
Components in Angular

- An angular application is a tree of Components
- The top level component is the application itself, which is rendered by the browser when bootstrapping the application.
- Components are:
  - Composable
  - Reusable
  - Hierarchical
- Let's take as an example an inventory management application

Inventory App: Components

Components in Angular

Navigation Component



Inventory Management App

Home | Products | Help

Products > Products List

SKU# 104544-2 $109.99
Nykee Running Shoes
Men > Shoes > Running Shoes

SKU# 187611-0 $238.99
South Face Jacket
Women > Apparel > Jackets & Vests

SKU# 443102-9 $238.99
Adeeds Active Hat
Men > Accessories > Hats

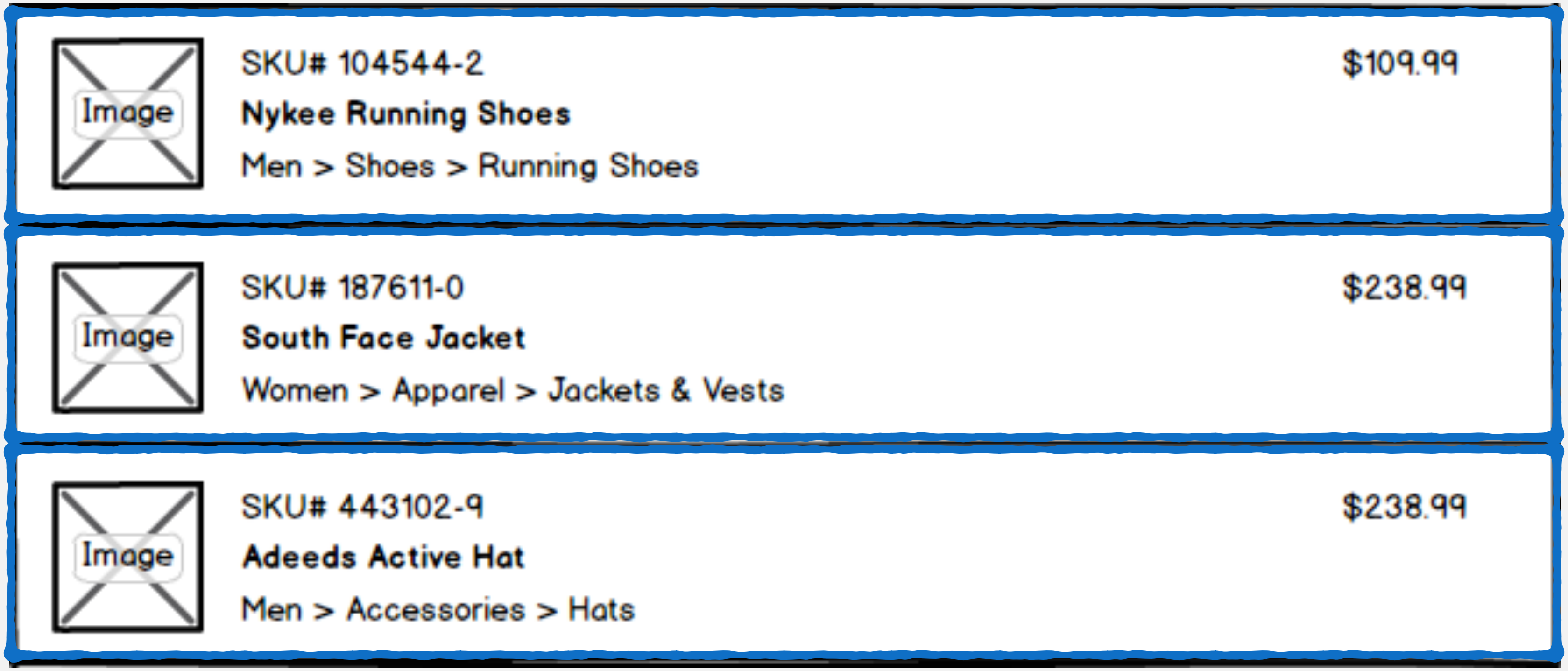
Breadcrumbs Component

ProductList Component

Inventory App: Components

Components in Angular

Product Row
Component



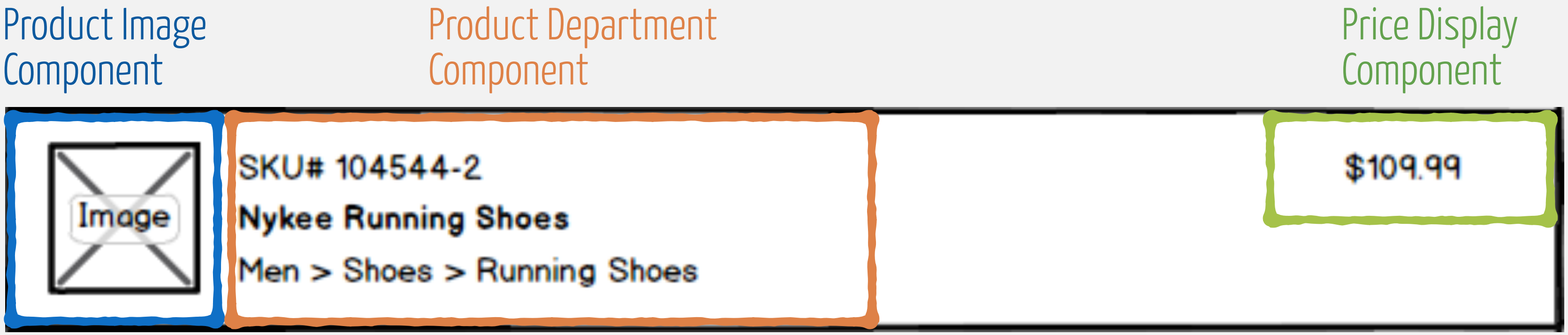
SKU# 104544-2 $109.99
Nykee Running Shoes
Men > Shoes > Running Shoes

SKU# 187611-0 $238.99
South Face Jacket
Women > Apparel > Jackets & Vests

SKU# 443102-9 $238.99
Adeeds Active Hat
Men > Accessories > Hats

Inventory App: Components

Components in Angular



Product Image Component

Product Department Component

Price Display Component

SKU# 104544-2

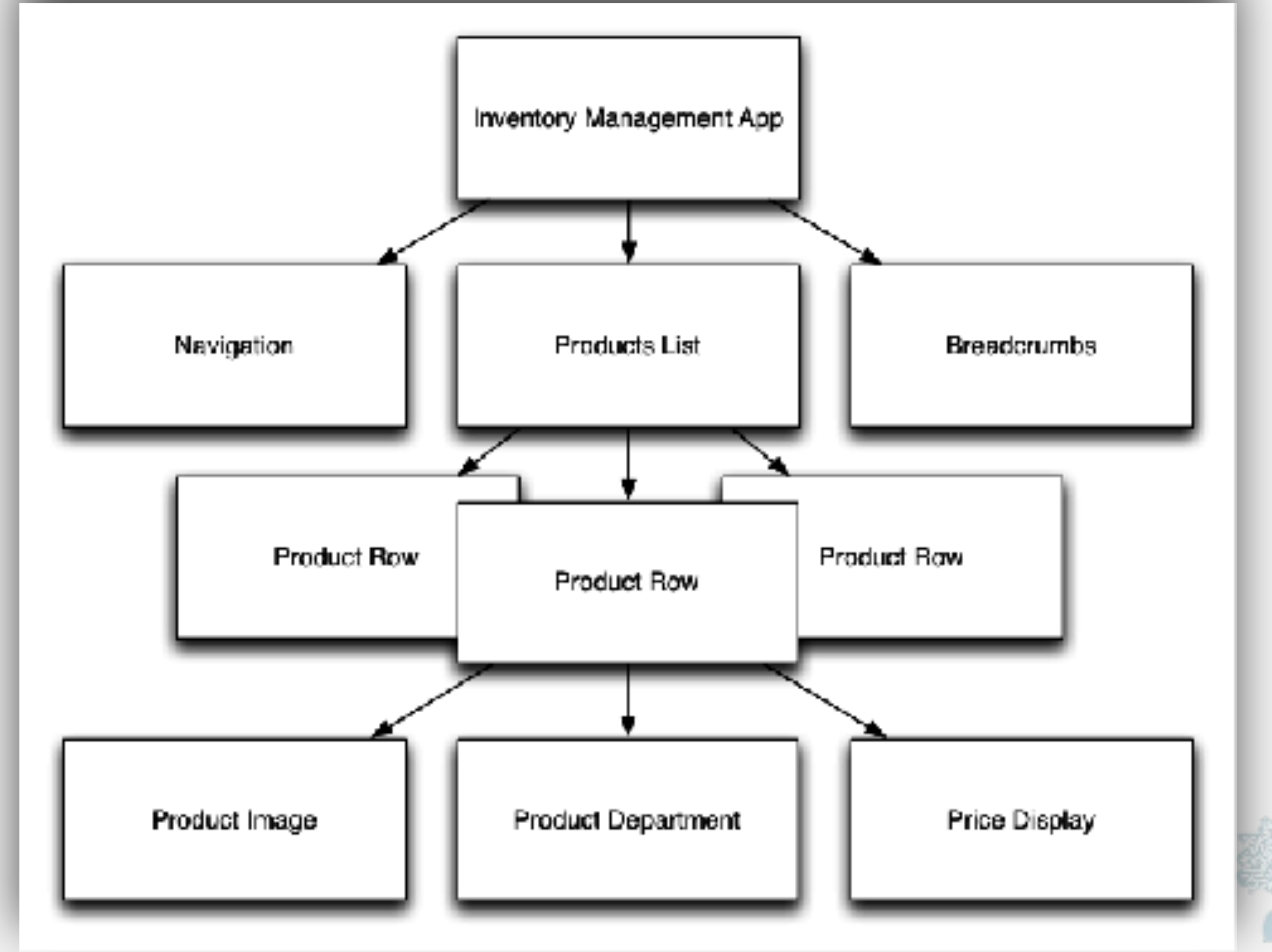
Nykee Running Shoes

Men > Shoes > Running Shoes

$109.99

Inventory App: Tree Representation

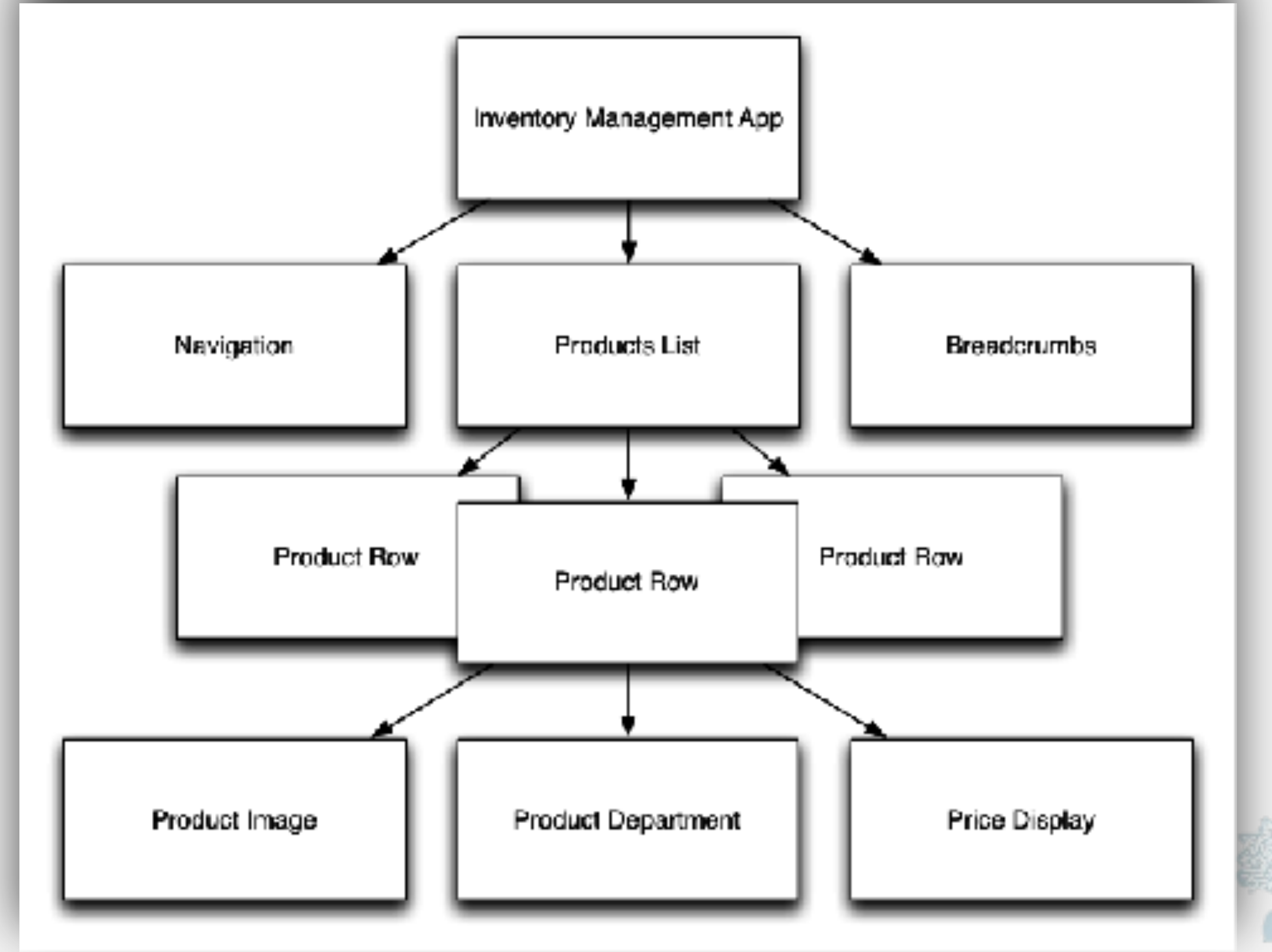
Components in Angular



MedTech

Inventory App: Tree Representation

Components in Angular



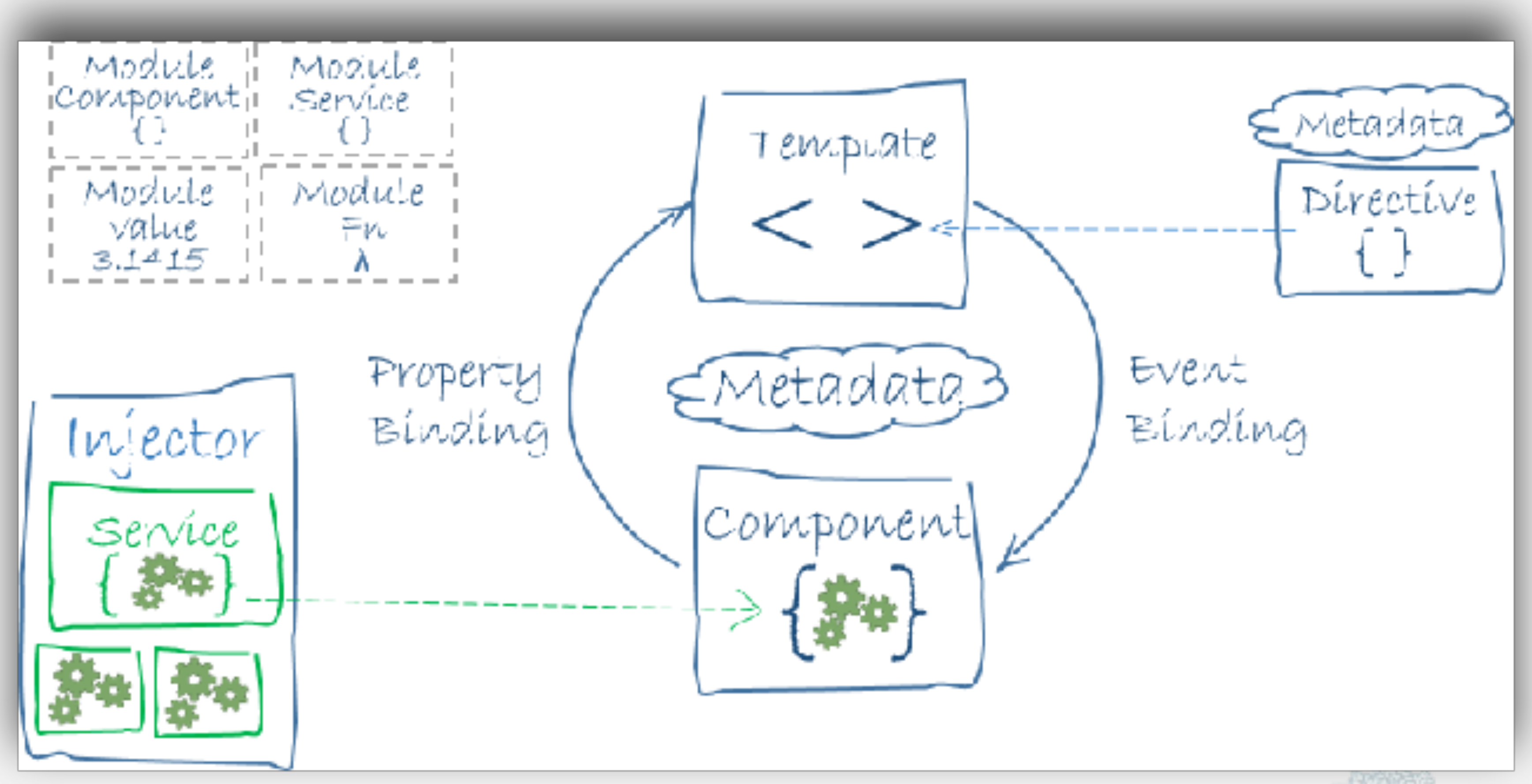
Angular

ANGULAR ARCHITECTURE

MedTech

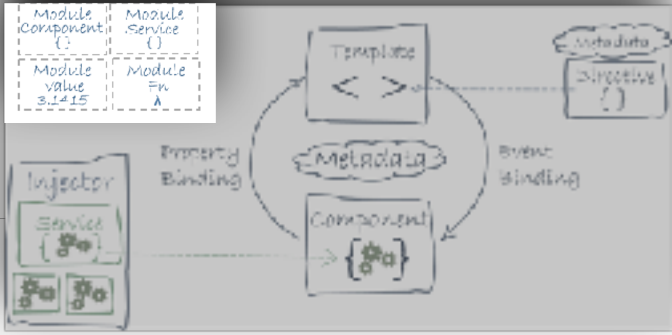
Architecture

Angular Architecture



Modules

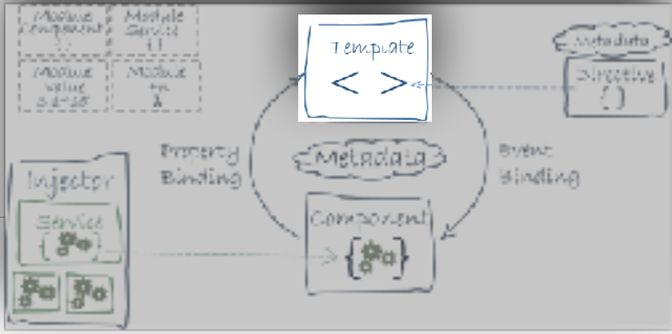
Angular Architecture



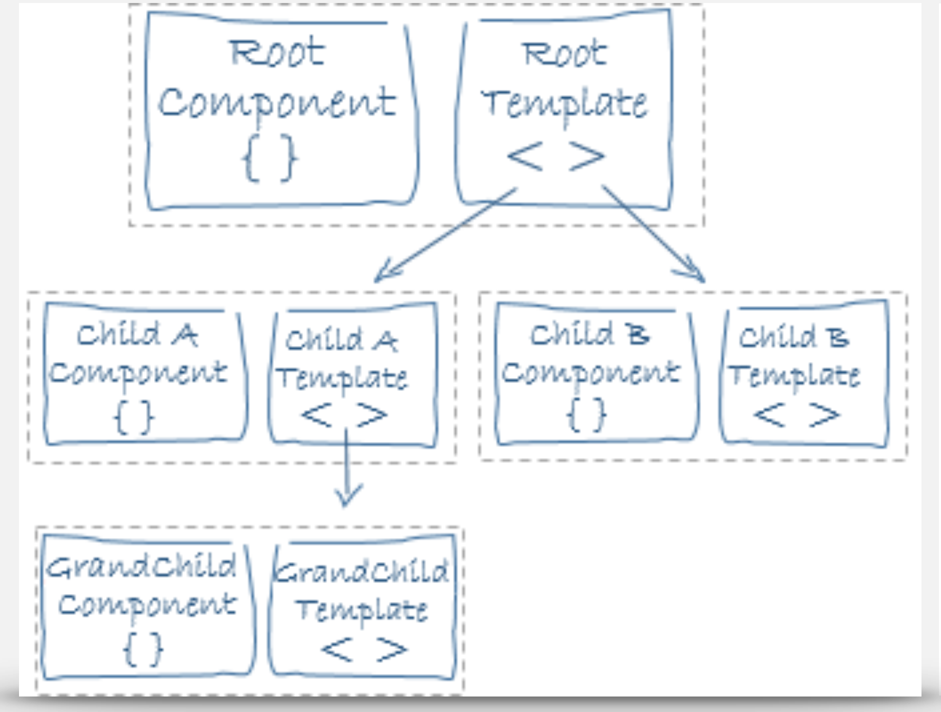
- Angular apps are modular:
  - An application defines a set of Angular Modules or NgModules
  - Every angular module is a class with an @NgModule decorator
- Every Angular App has at least one module: the root module
- There are other feature modules
  - Cohesive blocks of code, dedicated to an application domain, a workflow or a closely related set of capabilities
- NgModule takes a single metadata object describing the module, with the following properties
  - Declarations: view classes (components, directives and piped)
  - Exports: subset of public declarations, usable in the templates of other modules
  - Imports: external modules needed by the templates of this module
  - Providers: creators of services that this module contributes to
  - Bootstrap: main application view, called the root component, that hosts all other app views

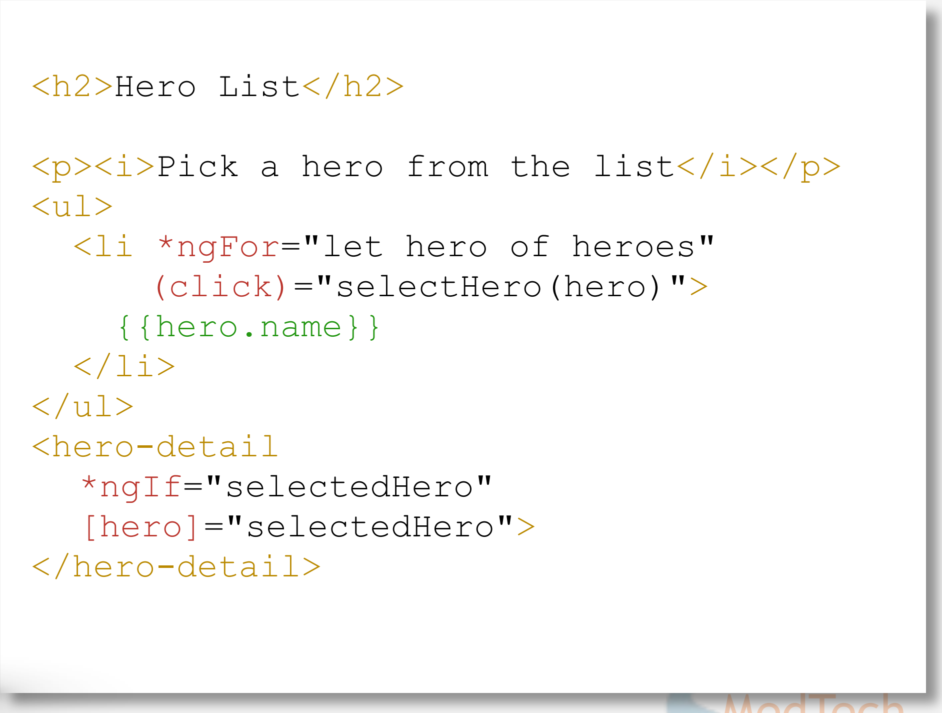
Templates

Angular Architecture



- A snippet of the HTML code of a component
  - A component's view is defined with its template
- Uses Angular's template syntax, with custom elements

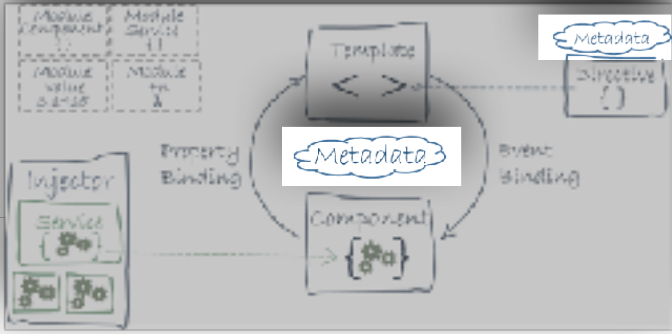




```
<h2>Hero List</h2>

<p><i>Pick a hero from the list</i></p>
<ul>
  <li *ngFor="let hero of heroes"
      (click)="selectHero(hero)">
    {{hero.name}}
  </li>
</ul>
<hero-detail
   *ngIf="selectedHero"
   [hero]="selectedHero">
</hero-detail>
```

Metadata

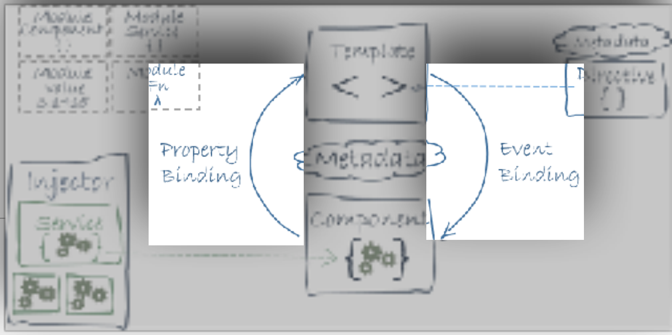
Angular Architecture



- Tells Angular how to process a class
- Uses decorators to attach information to a class:
  - @Component: identifies the class below it as a component class, with options:
    - moduleId: source of the base address (module.id) for module-relative URLs (such as templateURL)
    - selector: CSS selector for the template code
    - templateURL: address of the component's HTML template
    - providers: array of dependency injection providers for services that the component requires
  - Other metadata decorators:
    - @Injectable, @Input, @Output,..

Data Binding

Angular Architecture

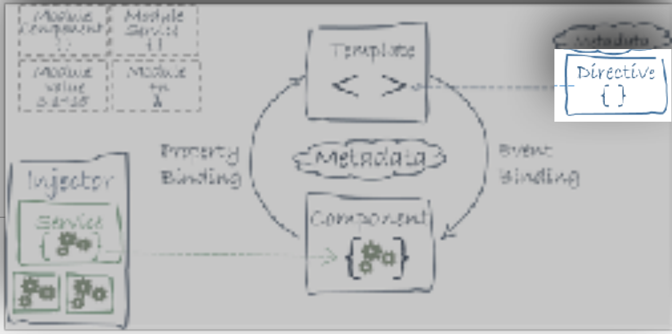


- Angular supports Data Binding
  - Mechanism for coordinating parts of a template with parts of a component
- Four main forms:
  - {{hero.main}}: interpolation
    - Displays the component's hero.name property value within the <li> element
  - [hero]: property binding
    - Passes the value of selectedHero to the child comp.
  - (click): event binding
    - Calls the component's selectHero method when the user clicks a hero's name
  - [(ngModel)]: Two-way data binding
    - Combines property and event binding, with ngModel

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

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

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

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

Directives

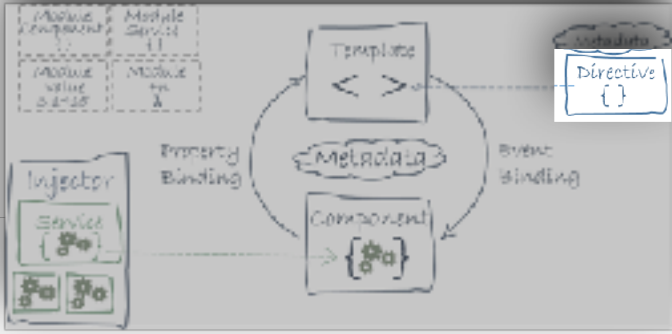
Angular Architecture



- Angular templates are dynamic
  - When Angular renders them, it transforms the DOM according to instructions given by directives
- A directive is a class with the @Directive decorator
- A component is a directive-with-a-template
  - A @Component decorator is actually a @Directive extended with template-oriented features
- Appear within an element tag as attributes do
- Two types of directives
  - Structural directives
  - Attribute directives

Directives

Angular Architecture



- Structural directives
  - Alter the layout by adding, removing and replacing elements in the DOM

```
<li *ngFor="let hero of heroes"></li>

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

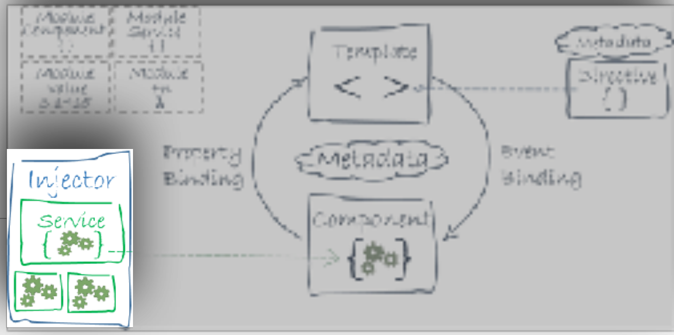
- Attribute directives
  - Alter the appearance or behaviour of an existant element
  - Look like regular HTML attributes

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

- Custom attributes
  - You can write your own directives

Services

Angular Architecture



- Almost anything can be a service
- A class with a narrow, well-defined purpose
  - Ex: logging servie, data service, tax calculator, application configuration,…
- There is no specific definition of a class in Angular, but classes are fundamental to any Angular application
- Component classes should be lean
  - They shouldn't fetch data from the server, validate user input or log directly to the console
  - They just deal with user experience, mediate between the view and the logic
  - Everything non trivial should be delegated to services
- A service is associated to a component using dependency injection

Angular

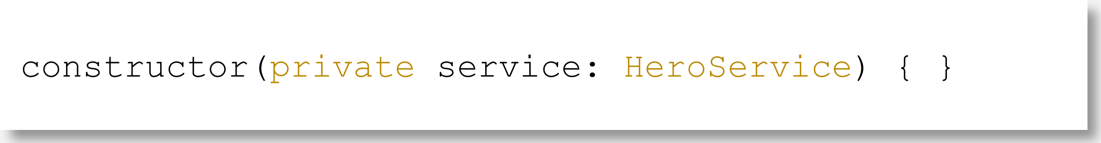
DEPENDENCY INJECTION

MedTech

Definition

Dependency Injection

- Important application design pattern
- Commonly called DI
- A way to supply a new instance of a class with the fully-formed dependencies it requires
- Most dependencies are services
  - DI is used to provide new components with the services they need
  - It knows which services to instantiate by looking at the types of the component's constructor parameters



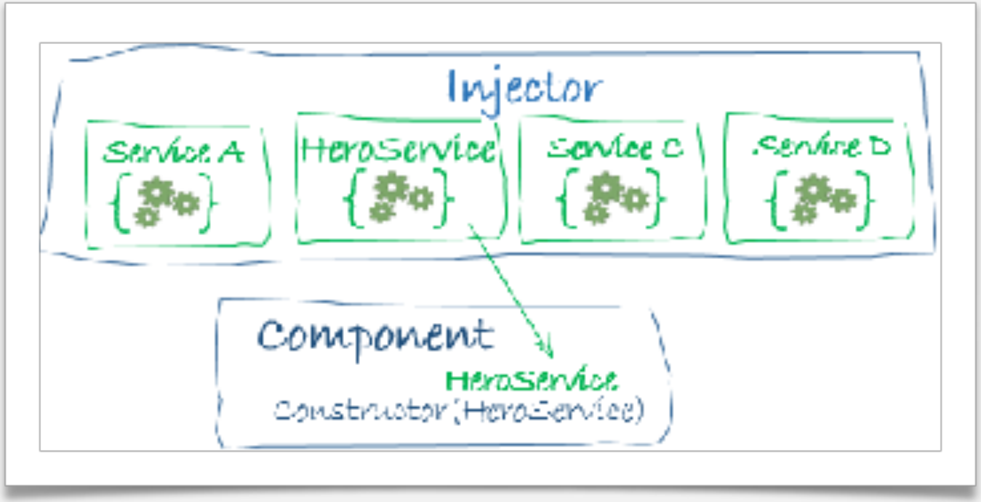
```
constructor(private service: HeroService) { }
```

- When Angular creates a component, it asks an injector for the services it requires

Injector

Dependency Injection

- Maintains a container of service instances that it has previously created
- If a requested service instance is not in the container, the injector makes one and adds it to the container before returning the service to Angular
- When all requested services have been resolved and returned, Angular can call the component's constructor with those services as arguments



Provider

Dependency Injection

- In order for the injector to know which services to instantiate, you need to register a provider of each one of them
- Provider: Creates or returns a service
- It is registered in a module or a component
  - Add it to the root module for it to be available everywhere
  - Register it in the component to get a new instance of the service with each new instance of the component

```
@NgModule({
  imports: [
    …
  ],
  providers: [
    HeroService,
    Logger
  ],
  …
})
```

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

@Injectable()

Dependency Injection

- @Injectable() marks a class as available to an injector for instantiation
- It is mandatory if the service class has an injected dependency
  - For example: if the service needs another service, which is injected in it
- It is highly recommended to add an @Injectable() decorator for every service class for the sake of
  - Future proofing
  - Consistency
- All components and directives are already subtypes of Injectable
  - Even though they are instantiated by the injector, you don't have to add the @Injectable() decorator to them

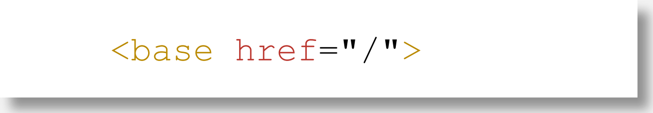
Angular

ROUTING

Angular Router

Routing

- Enables navigation from one view to the next as users perform application tasks
- Interprets a browser URL as an instruction to navigate to a client-generated view
- Can pass optional parameters to the supporting view component to help it decide which specific content to display
- Logs activity in the browser's history journal so the back and forward buttons work
- Most routing applications add a <base> element to the index.html as the first child of <head>
    - Tells the router how to compose navigation URLs

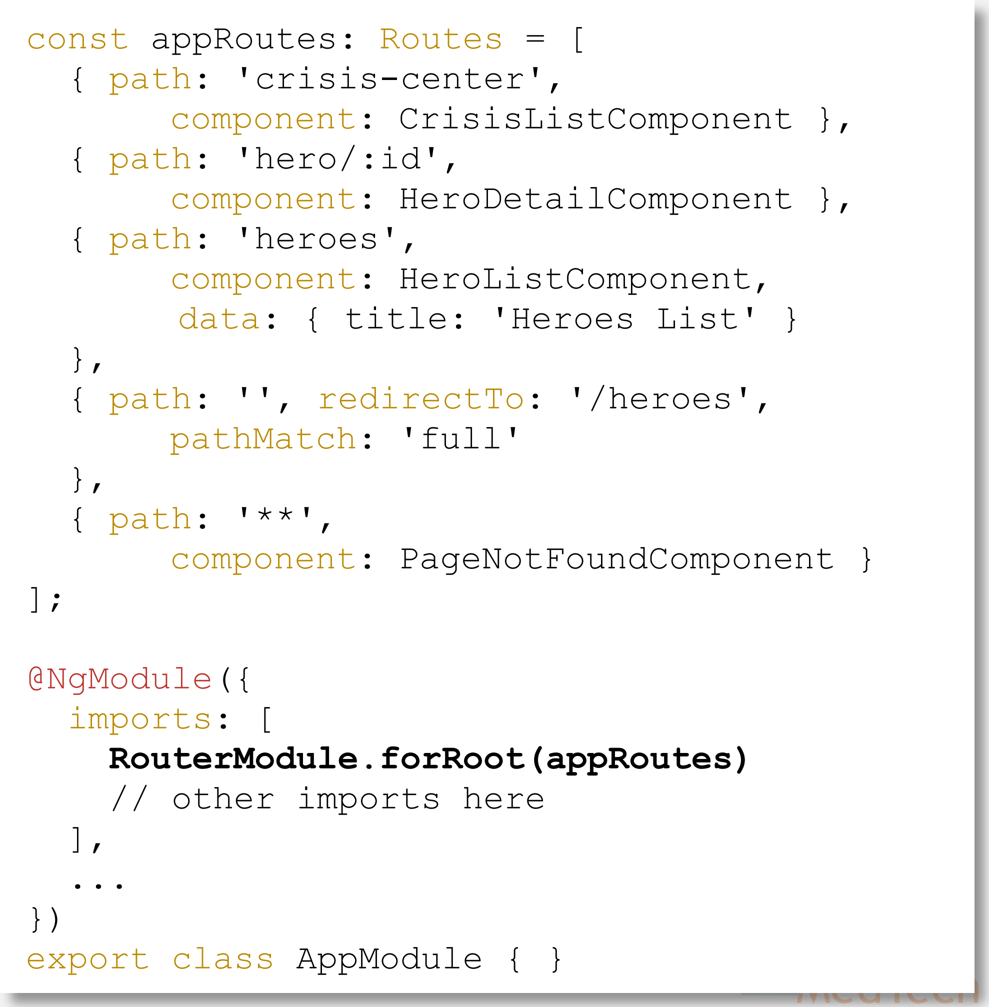


```
<base href="/">
```

Angular Router

Routing

- One singleton instance of the Router service exists for an application
- When the browser's URL changes, that router looks for the corresponding Route to know which component to display
- A router has no routes until you configure it
  - Using the RouterModule.forRoot method

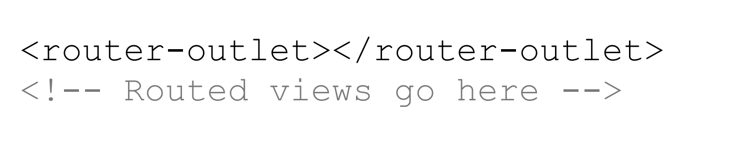


```
const appRoutes: Routes = [
  { path: 'crisis-center',
       component: CrisisListComponent },
  { path: 'hero/:id',
       component: HeroDetailComponent },
  { path: 'heroes',
       component: HeroListComponent,
        data: { title: 'Heroes List' }
  },
  { path: '', redirectTo: '/heroes',
       pathMatch: 'full'
  },
  { path: '**',
       component: PageNotFoundComponent }
];

@NgModule({
  imports: [
    RouterModule.forRoot(appRoutes)
    // other imports here
  ],
  ...
})
export class AppModule { }
```

Router Views

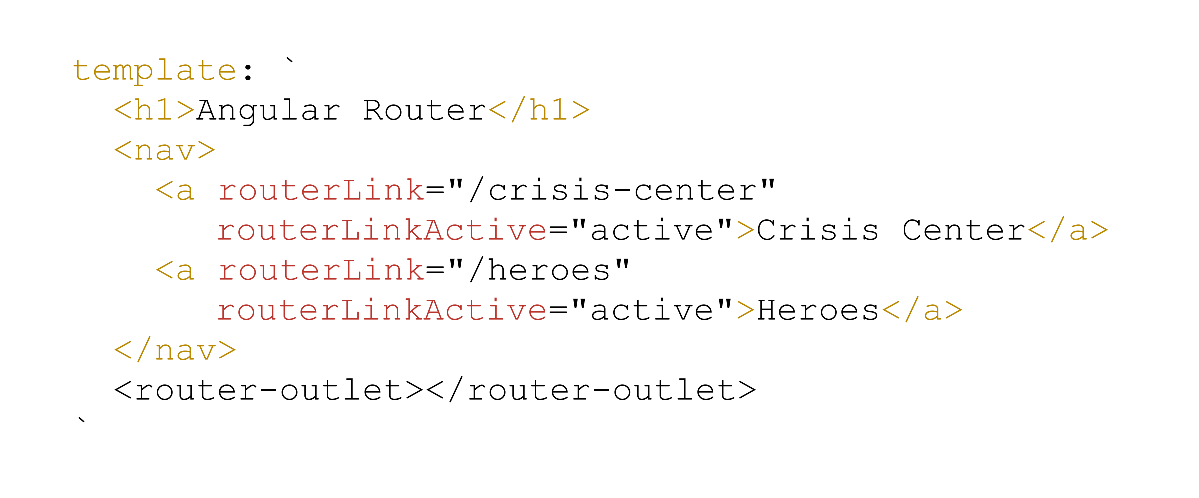
Routing

- In order to render the component chosen by the router, a RouterOutlet is inserted in the template



```
<router-outlet></router-outlet>
<!-- Routed views go here -->
```

- To navigate from a route to another, you use routerLinks
  - routerLinkActive associates a CSS class "active" to the cliqued link



```
template: `
  <h1>Angular Router</h1>
  <nav>
    <a routerLink="/crisis-center"
       routerLinkActive="active">Crisis Center</a>
    <a routerLink="/heroes"
       routerLinkActive="active">Heroes</a>
  </nav>
  <router-outlet></router-outlet>
`
```

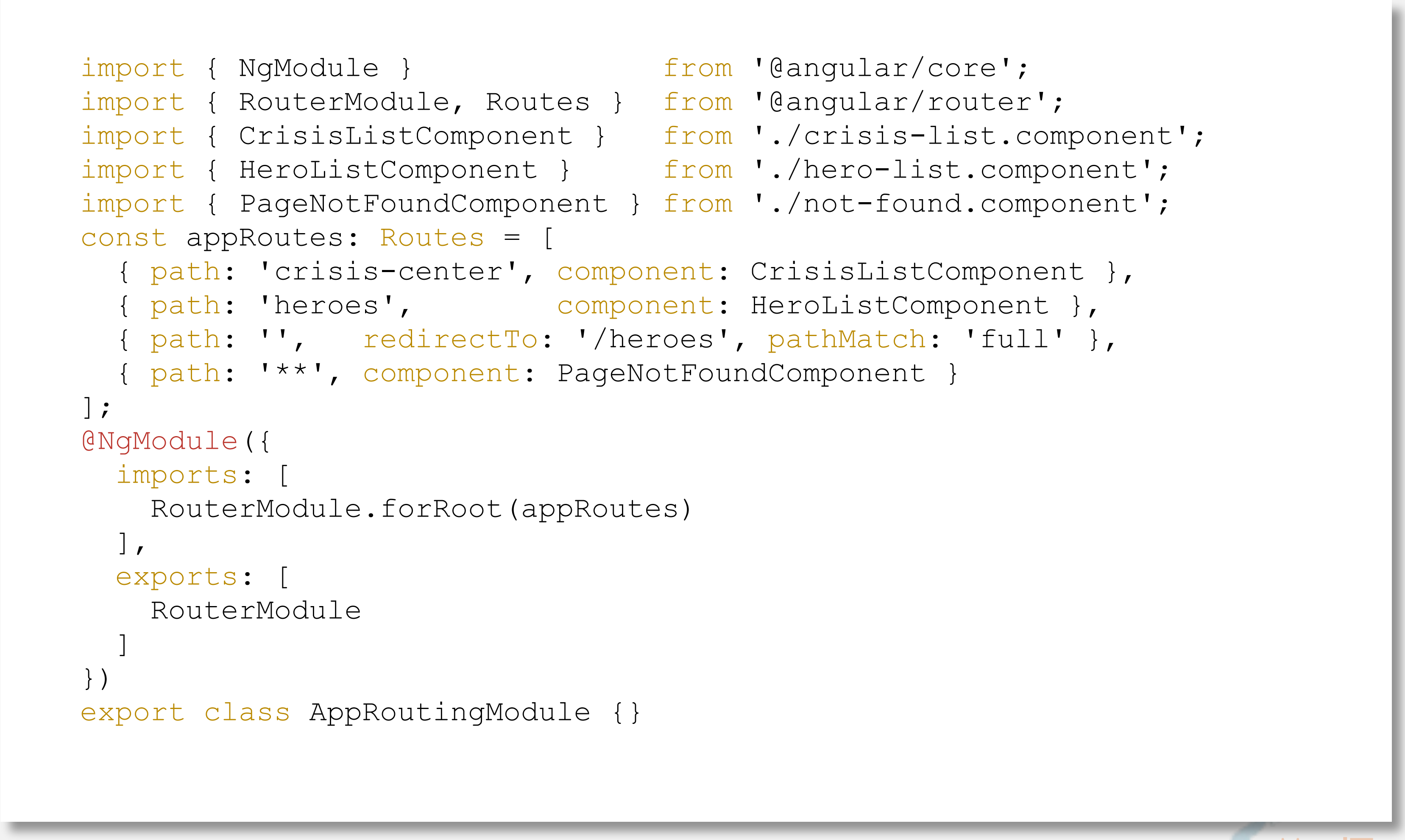
Routing Module

Routing

- For simple routing, defining the routes in the main application module is fine
- It can become more difficult to manage if the application grows and you use more Router features
    - Refactor the routing configuration in its own file: the Routing Module
- The Routing Module
    - Separates routing concerns from other application concerns
    - Provides a module to replace or remove when testing the application
    - Provides a well-known location for routing service providers
    - Does not declare components

Routing Module: Example

Routing



```
import { NgModule }              from '@angular/core';
import { RouterModule, Routes }  from '@angular/router';
import { CrisisListComponent }   from './crisis-list.component';
import { HeroListComponent }     from './hero-list.component';
import { PageNotFoundComponent } from './not-found.component';
const appRoutes: Routes = [
  { path: 'crisis-center', component: CrisisListComponent },
  { path: 'heroes',        component: HeroListComponent },
  { path: '',   redirectTo: '/heroes', pathMatch: 'full' },
  { path: '**', component: PageNotFoundComponent }
];
@NgModule({
  imports: [
    RouterModule.forRoot(appRoutes)
  ],
  exports: [
    RouterModule
  ]
})
export class AppRoutingModule {}
```

Navigation Guards

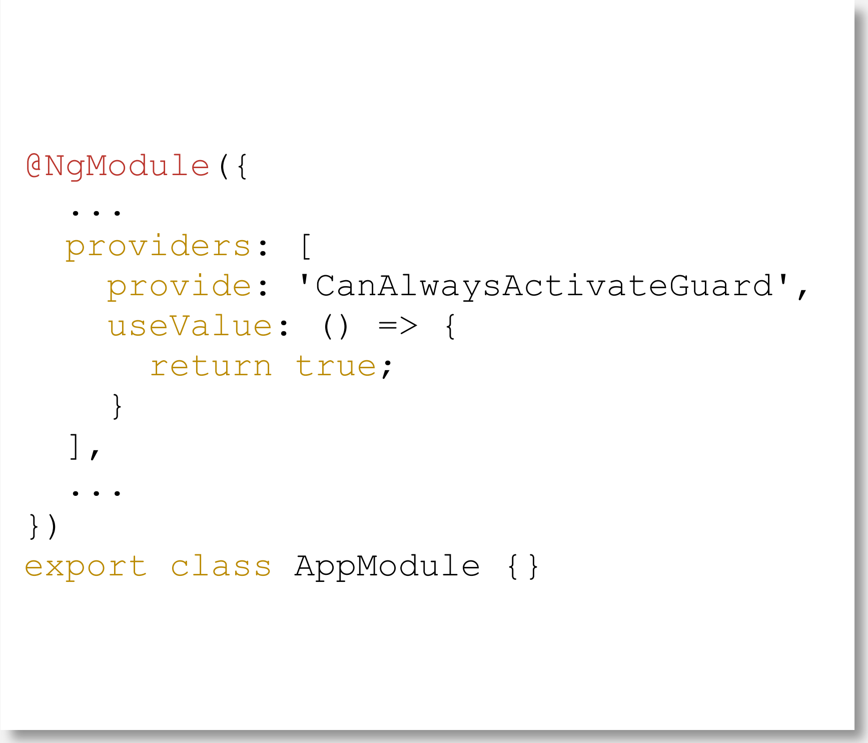
Routing

- Sometimes, routes need to be protected:
  - to prevent users from accessing areas that they're not allowed to access
  - to ask for permission, …
- Navigation Guards are applied to routes to do that
- Four guard types:
  - CanActivate: decides if a route can be activated
  - CanActivateChild: decides if child routes of a route can be activated
  - CanDeactivate: decides if a route can be deactivated
  - CanLoad: decides if a module can be loaded lazily
- Guards can be implemented in different ways, but mainly, you obtain a function that returns Observable<boolean>, Promise<boolean> or boolean

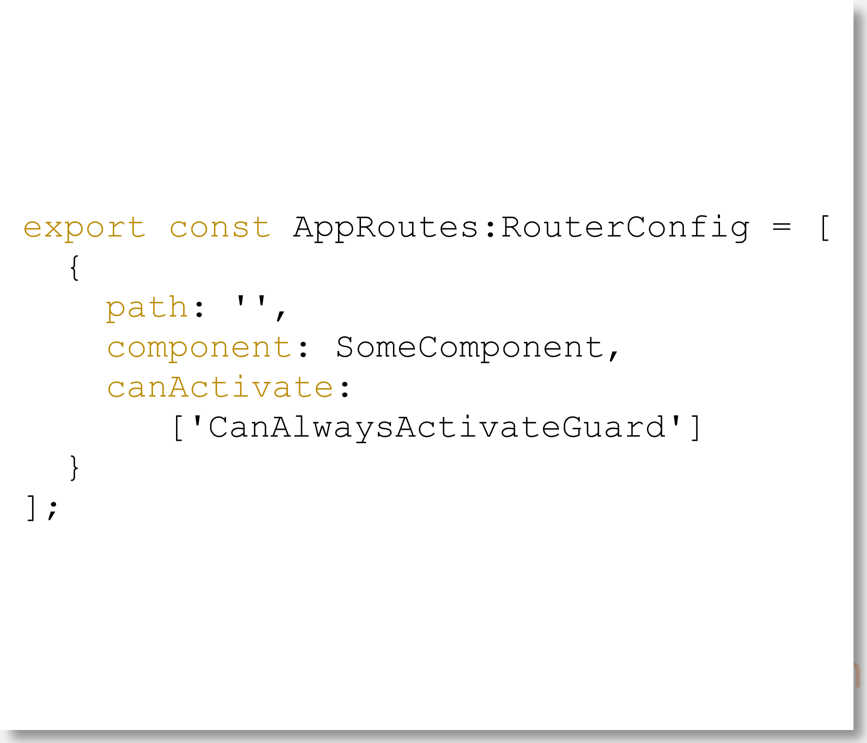
Navigation Guards: as Functions

Routing

- To register a guard as a function, you need to define a token and the guard function, represented as a provider
- Once the guard registered with a token, it is used on the route configuration



```
@NgModule({
  ...
  providers: [
    provide: 'CanAlwaysActivateGuard',
    useValue: () => {
      return true;
    }
  ],
  ...
})
export class AppModule {}
```



```
export const AppRoutes:RouterConfig = [
  {
    path: '',
    component: SomeComponent,
    canActivate:
       ['CanAlwaysActivateGuard']
  }
];
```

Navigation Guards: as Classes

Routing

- Sometimes, a guard needs DI capabilities
  - Should be declared as Injectable classes
- Implement in this case CanActivate, CanDeactivate or CanActivateChild interfaces

```
import { Injectable } from '@angular/core';
import { CanActivate } from '@angular/router';
import { AuthService } from './auth.service';

@Injectable()
export class CanActivateViaAuthGuard
       implements CanActivate {

  constructor(private authService: AuthService){
   }

  canActivate() {
    return this.authService.isLoggedIn();
  }
```

```
@NgModule({
  ...
  providers: [
    AuthService,
    CanActivateViaAuthGuard
  ]
})
export class AppModule {}


…

{
  path: '',
  component: SomeComponent,
  canActivate: [
    'CanAlwaysActivateGuard',
     CanActivateViaAuthGuard
  ]
}
```

References

- Sites
  - Angular2 official documentation, https://angular.io/docs, consulted in March 2017
  - Pascal Precht, Protecting Routes Using Guards in Angular, https://blog.thoughtram.io/angular/2016/07/18/guards-in-angular-2.html#as-classes, updated in December 2016, consulted in March 2017
- Textbook
  - Rangle's Angular2 Training Book, rangle.io, Gitbook
  - Ang-book 2, the complete book on AngularJS2, 2015-2016