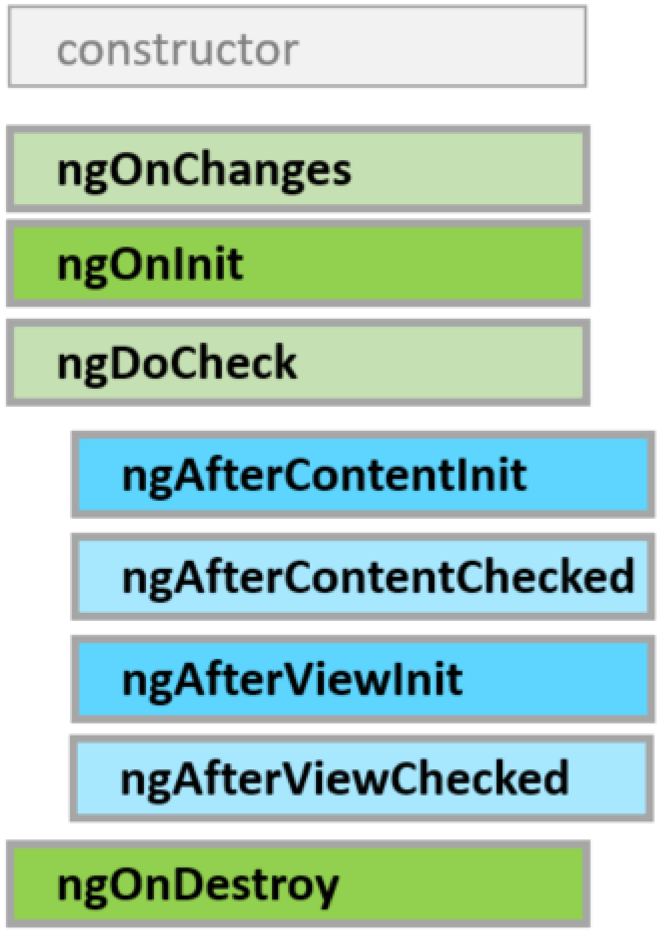
Lifecyle hooks

Component Lifecycle Hooks

- A component has a lifecycle managed by Angular.
- Angular creates it, renders it, creates and renders its children, checks it when its data-bound properties change, and destroys it before removing it from the DOM.
- Angular offers lifecycle hooks that provide visibility into these key life moments and the ability to act when they occur.

After Angular creates a component/directive by calling `new` on its constructor, it calls the lifecycle hook methods in the following sequence at specific moments.

Angular only calls a directive/component hook method if it is defined.



Lifecycle Example

This component will be notified when its input properties change.

```
@Component({
        selector: 'app-cmp'
})
class AppCmp implements OnChanges {
        @Input() field1;
        @Input() field2;

        ngOnChanges(changes) {
                //..
        }
}
```

Angular Forms

Angular Forms

- Angular forms are used to handle user's input.

- Angular Form building blocks:
  - FormControl: It tracks the value and validation status of the individual form control.
  - FormGroup: It tracks the same values and status for the collection of form controls.
  - FormArray:It tracks the same values and status for the array of the form controls.
  - ControlValueAccessor: It creates a bridge between Angular FormControl instances and native DOM elements.

Two Types of Forms

Template driven Forms	Reactive Forms
Created by Directive	Created in Component
Good for Simple forms	More Flexible, more complex scenarios
Simple Validation	More control over validation logic
Unit untestable (almost)	Unit testable
In `FormsModule`	In `ReactiveFormsModule`

Template-driven Forms

Template driven Forms

- Template drive Froms make use of built-in directives to build forms such as `ngModel`, `ngModelGroup` and `ngForm` available from `FormsModule` module
- With a template driven form, most of the work is done in the template
- We need to import `FromsModule` in `app.module.ts`

```
import { FormsModule } from '@angular/forms';
import { BrowserModule } from '@angular/platform-browser';

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

Template Driven Form Directives

- `ngForm` **directive**
  - It holds the controls you created for the elements with an `ngModel` directive and name attribute, and monitors their properties, including their validity.
  - It also has its own valid property which is true only if every contained control is valid.

- `ngModel` **directive**
  - Creates a `FormControl` instance from a domain model and binds it to a form control element.
  - The `NgModel` directive allows you to display a data property and update that property when the user makes changes.
  - `[(ngModel)]`: Two-way binding

Create Form

```
<form #heroForm="ngForm" (ngSubmit)="onSubmit(heroForm.value)">
        <div class="form-group">
            <label for="name">Name</label>
            <input class="form-control" id="name" required [(ngModel)]="model.name" name="name">
        </div>

        <div class="form-group">
            <label for="alterEgo">Alter Ego</label>
            <input class="form-control" id="alterEgo" [(ngModel)]="model.alterEgo" name="alterEgo">
        </div>
        <div class="form-group">
            <label for="power">Hero Power</label>
            <select class="form-
control" id="power" required [(ngModel)]="model.power" name="power">
                <option *ngFor="let pow of powers" [value]="pow">{{pow}}</option>
            </select>
        </div>

        <button type="submit" class="btn btn-success">Submit</button>

    </form>
```

Component

```
@Component({
  selector: 'app-hero-form',
  templateUrl: './hero-form.component.html',
  styles: []
})
export class HeroFormComponent {

  powers: string[] = ['Really Smart', 'Super Flexible',
    'Super Hot', 'Weather Changer'];

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

  submitted = false;

  onSubmit(f) {
    this.submitted = true;
    console.log(f.value);
  }
}
```

Validation - Track control state and validity with *ngModel*

- The *NgModel* directive tracks state;
- It also updates the control with special Angular CSS classes that reflect the state.

State	**Class if true**	**Class if false**
The control has been visited.	ng-touched	ng-untouched
The control's value has changed.	ng-dirty	ng-pristine
The control's value is valid.	ng-valid	ng-invalid

Validation - Show and hide validation error messages

- A template reference variable e.g. #firstName
- The "is required" message in a nearby <div>, which you'll display only if the control is invalid.

```
<form #heroForm="ngForm" (ngSubmit)="onSubmit(heroForm)">
        <div class="form-group">
            <label for="name">Name</label>

            <input class="form-control" id="name" required
                 [(ngModel)]="model.name" name="name" #name="ngModel">
            <div [hidden]="name.valid || name.pristine" class="alert alert-danger">
                Name is required
            </div>
        </div>
        <button type="submit" class="btn btn-success" [disabled]="!heroForm.form.valid">Submit</button>
</form>
```

https://angular.io/api/forms/NgModel

Reactive Forms

Reactive Forms

- Create forms using `FormControl`, `FormGroup` and `FromBuilder` from `ReactiveFormsModule` module.
- We need to import `ReactiveFormsModule` in `app.module.ts`

```
import { ReactiveFormsModule } from '@angular/forms';
import { BrowserModule } from '@angular/platform-browser';

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

Reactive forms API

- Listed below are the base classes and services used to create and manage form controls.

Class	Description
AbstractControl	The abstract base class for the concrete form control classes FormControl, FormGroup, and FormArray. It provides their common behaviors and properties.
FormControl	Manages the value and validity status of an individual form control. It corresponds to an HTML form control such as <input> or <select>.
FormGroup	Manages the value and validity state of a group of AbstractControl instances. The group's properties include its child controls. The top-level form in your component is FormGroup.
FormArray	Manages the value and validity state of a numerically indexed array of AbstractControl instances.
FormBuilder	An injectable service that provides factory methods for creating control instances.

Create Form

- Form model: represent the form as a *model* composed of instances of FormGroups and FormControls, or use FormBuilder

```
export class ProfileEditorComponent {

  profileForm = new FormGroup({
    firstName: new FormControl('', Validators.required),
    lastName: new FormControl('', Validators.required),
    address: new FormGroup({
      street: new FormControl(''),
      city: new FormControl(''),
      state: new FormControl(''),
      zip: new FormControl('')
    })
  });

  onSubmit() {
    console.log(this.profileForm.value);
  }
}
```

Create Form use FormBuilder

- The FormBuilder is a more powerful way that helps us build forms.
- Forms are made up of FormControl and FormGroup and the FormBuilder helps us create them (you can think of it as a factory object).

```
export class ProfileEditorComponent {

  constructor(private fb: FormBuilder) { }

  profileForm = this.fb.group({
    firstName: ['', Validators.required],
    lastName: ['', Validators.required],
    address: this.fb.group({
      street: [''],
      city: [''],
      state: [''],
      zip: ['']
    })
  });
```

HTML Form

- Linking the form model to the form template

```
<form [formGroup]="profileForm" (ngSubmit)="onSubmit()">

        <div class="form-group">
            <label>
                First Name:
                <input type="text" formControlName="firstName">
            </label>
            <div *ngIf="profileForm.controls.firstName.touched && profileForm.get('firstName').hasError('required')">
                First Name is required
            </div>
        </div>
        <div class="form-group">
            <label>
                Last Name:
                <input type="text" formControlName="lastName">
            </label>
            <div *ngIf="profileForm.controls.lastName.touched && profileForm.get('lastName').hasError('required')">
                Last Name is required
            </div>
        </div>
```

HTML Form (Cont.)

```
    <div formGroupName="address">
             <h3>Address</h3>

             <label>Street:<input type="text" formControlName="street"></label>

             <label>City:<input type="text" formControlName="city"></label>

             <label>State:<input type="text" formControlName="state"></label>

             <label>Zip Code:<input type="text" formControlName="zip"></label>
     </div>

     <button type="submit" [disabled]="!profileForm.valid">Submit</button>
</form>
```

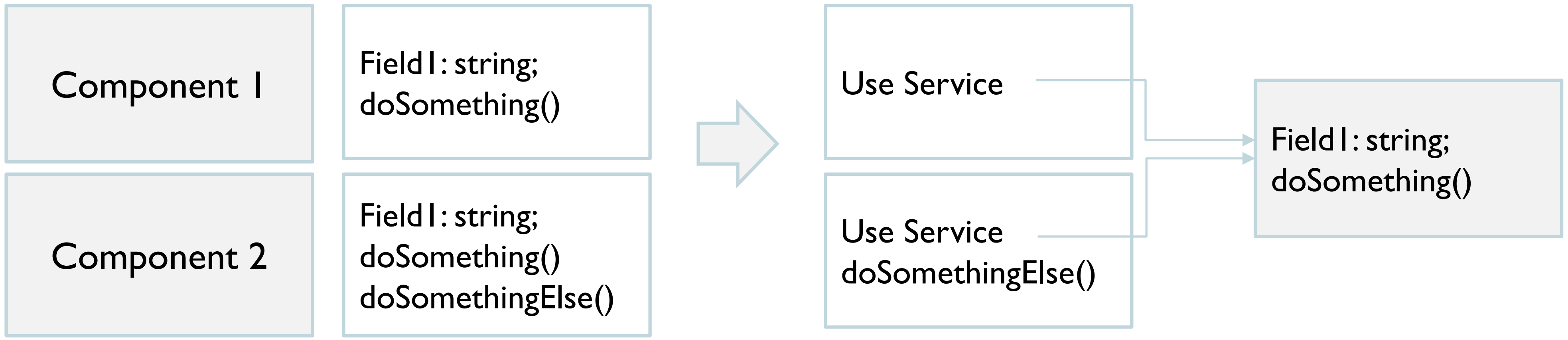
Dependency Injection

Dependency Injection

- Dependencies are services or objects that a class needs to perform its function.
- DI is a coding pattern in which a class asks for dependencies from external sources rather than creating them itself.

Angular Service

- Angular services are mostly useful in the following scenarios:
  - for separating the business logic of your app from the rendering logic in components,
  - for sharing the data between multiple components in your Angular app,
  - for easing testing and debugging,
  - for writing re-usable code.



Example

Let's imagine we have a `Product` class. Each product has a base price. In order to calculate the full price for this product, we rely on a service that takes the base price of the product when it's been initialized and calculate the price based the state we're selling it to.

```
class Product {
        constructor(basePrice: number) {
                this.service = new PriceService();
                this.basePrice = basePrice;
        }
        price(state: string) {
                return this.service.calculate(this.basePrice, state);
        }
}
```

A Better Way

```
class Product {
        constructor(service: PriceService, basePrice: number) {
                this.service = service;
                this.basePrice = basePrice;
        }
        price(state: string) {
                return this.service.calculate(this.basePrice, state);
        }
}
```

This technique of injecting the dependencies relies on a principle called the **Inversion of Control (IoC)** principle is also called informally the Hollywood principle (don't call us, we'll call you).

When we use DI we are moving towards a more **loosely coupled** architecture where changing bits and pieces of a single component affects the other areas of the application less. And, as long as the interface between those components don't change, we can even swap them altogether, without any other components even realizing.

Dependency Injection Parts

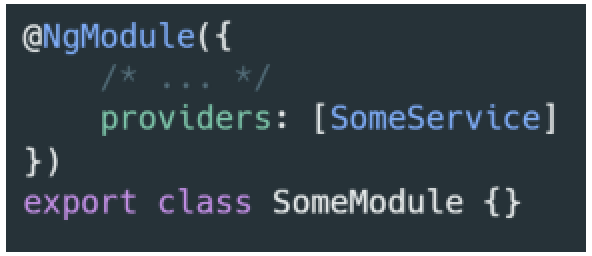
Dependency injection in Angular has three pieces:

- **The Provider** maps a token to a list of dependencies. It tells Angular how to **create/instantiate** an object.
- **The Injector** that holds a set of bindings and is responsible for resolving dependencies and injecting them when creating objects.
- **The Dependency/Injectable** that is what's being injected.

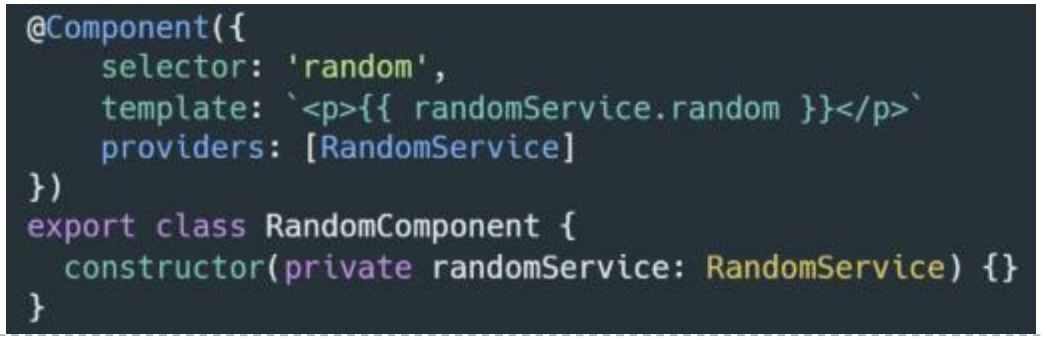
In Angular when you want to access an injectable you inject a dependency into a function and Angular's dependency injection framework will locate it and provide it to you.

Provider

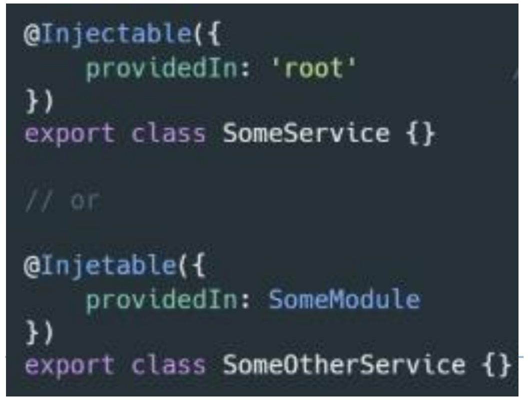
- In order for the injector to know which services to instantiate, you need to register a provide of each one of them
- Provider: Creates or returns a service
- It's registered in:
  - Add it to the root module for it be available everywhere
  - Register it in the component to get a new instance of the service with each new instance of the component
  - The Injectable decorator takes a property called provideIn that have by default the root value.



```
@NgModule({
    /* ... */
    providers: [SomeService]
})
export class SomeModule {}
```



```
@Component({
    selector: 'random',
    template: `<p>{{ randomService.random }}</p>`
    providers: [RandomService]
})
export class RandomComponent {
  constructor(private randomService: RandomService) {}
}
```



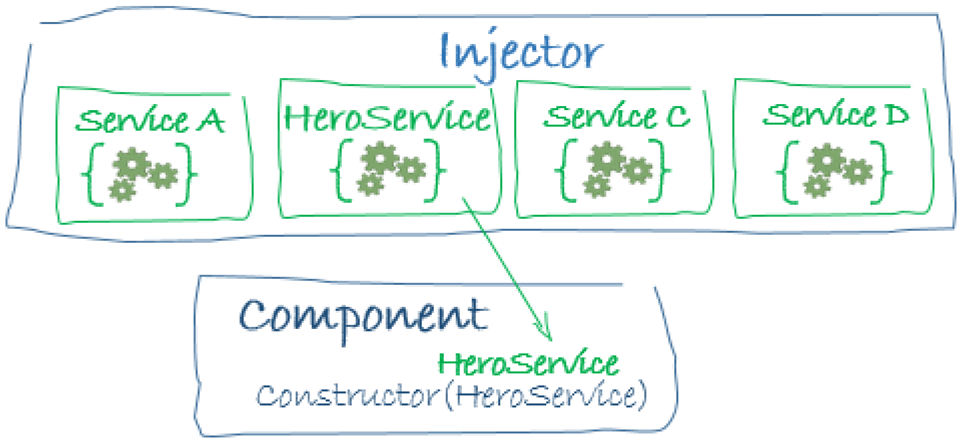
```
@Injectable({
    providedIn: 'root'
})
export class SomeService {}

// or

@Injetable({
    providedIn: SomeModule
})
export class SomeOtherService {}
```

Injector

- 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



Dependency Injection Algorithm

When resolving the backend dependency of a component, Angular will start with the injector of the component itself. Then, if it is unsuccessful, it will climb up to the injector of the parent component, and, finally, will move up to the injector created from AppModule.

```
let inj = this;
while (inj) {
   if (inj.has(requestedDependency)) {
      return inj.get(requestedDependency);
   } else {
       inj = inj.parent;
   }
}
throw new NoProviderError(requestedDependency);
```

Hierarchical Injector

Angular Dependency Injector is **hierarchical**. This basically means, that providers may be set up on different levels of the application, leading to different outcomes.

- All child components of a component will receive the **same service instance**, if the provider is set up on the parent component.
- If providers are specified on each individual child component, **different instances** will be created.
- The highest possible provider level, is the providers array in the `AppModule`.

@Injectable()

- @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
- 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

Create Services from CLI

- Services are **normal classes** and don’t receive any special metadata to become services.
- In order to use services, they need to be **injected** into the components/classes which should use them.
- To create a new service from Angular CLI:
  - `ng g service serviceName`

Injecting a Service

- Injecting a singleton instance of a class is probably the most common type of injection.

```
@Injectable({
  // we declare that this service
  //should be created
  // by the root application injector.
  providedIn: 'root'
})
export class HeroService {

  constructor() { }
  getHeroes() { return HEROES; }
}
```

```
@Component({
  selector: 'app-heros',
  templateUrl: './heros.component.html',
  styles: []
})
export class HerosComponent {
  heroes: Hero[];

  constructor(heroService: HeroService) {
        this.heroes = heroService.getHeroes();
  }
}
```

Inject a service into Another Service

```
@Injectable({
  // we declare that this service should be created
  // by the root application injector.
  providedIn: 'root'
})
export class HeroService {

  constructor(@Optional() private logger?: LoggerService) {
    if (this.logger) {
      this.logger.log('This is in hero service');
    }
  }
  getHeroes() { return HEROES; }
}
```

```
@Injectable({
  providedIn: 'root'
})
export class LoggerService {

  logs: string[] = [];

  log(message: string) {
    this.logs.push(message);
    console.log(message);
  }
}
```

References

- https://angular.io/guide/lifecycle-hooks

- https://angular.io/guide/reactive-forms
- https://angular.io/guide/forms
- https://angular.io/guide/form-validation

- https://angular.io/guide/dependency-injection