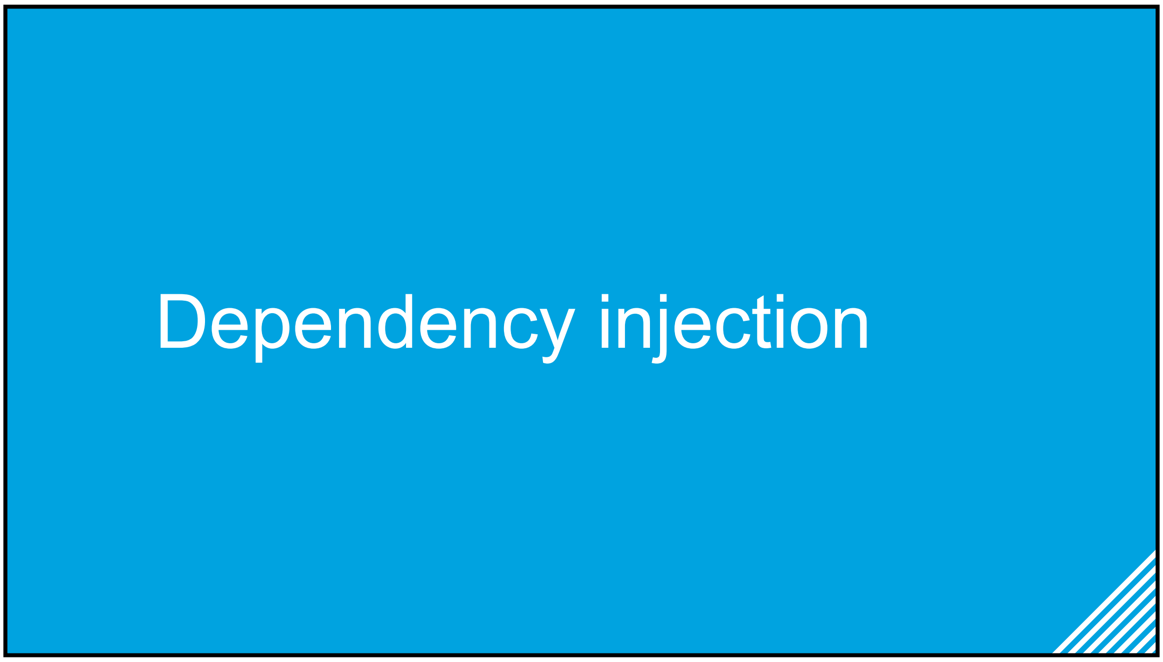
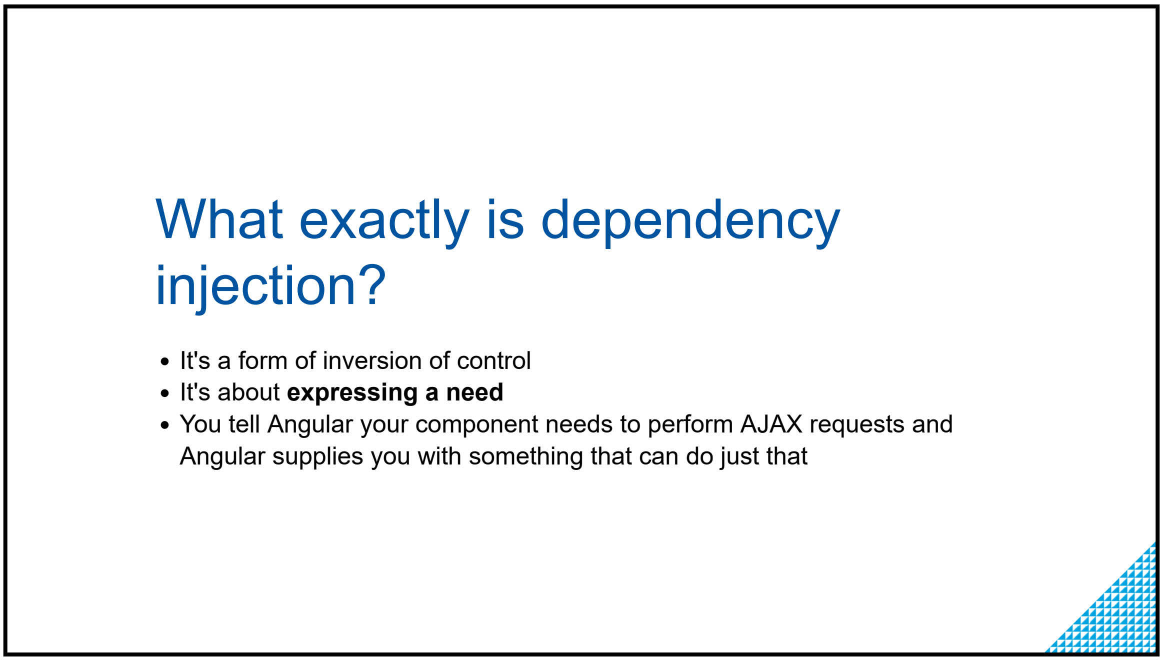
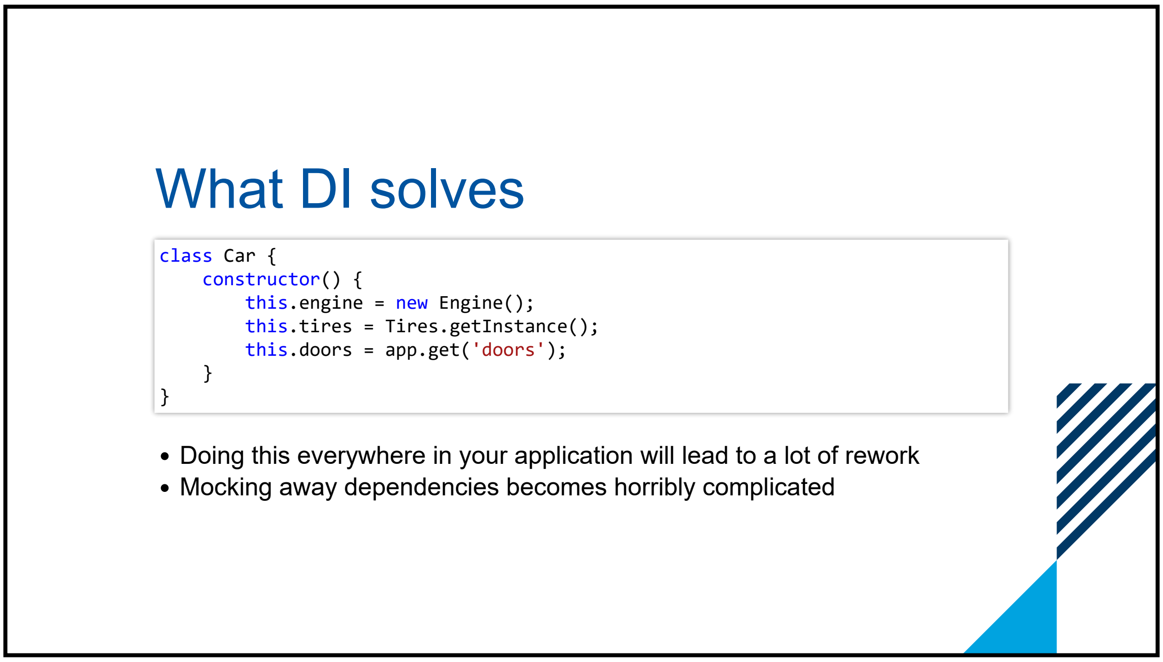


# Dependency injection



## What exactly is dependency injection?

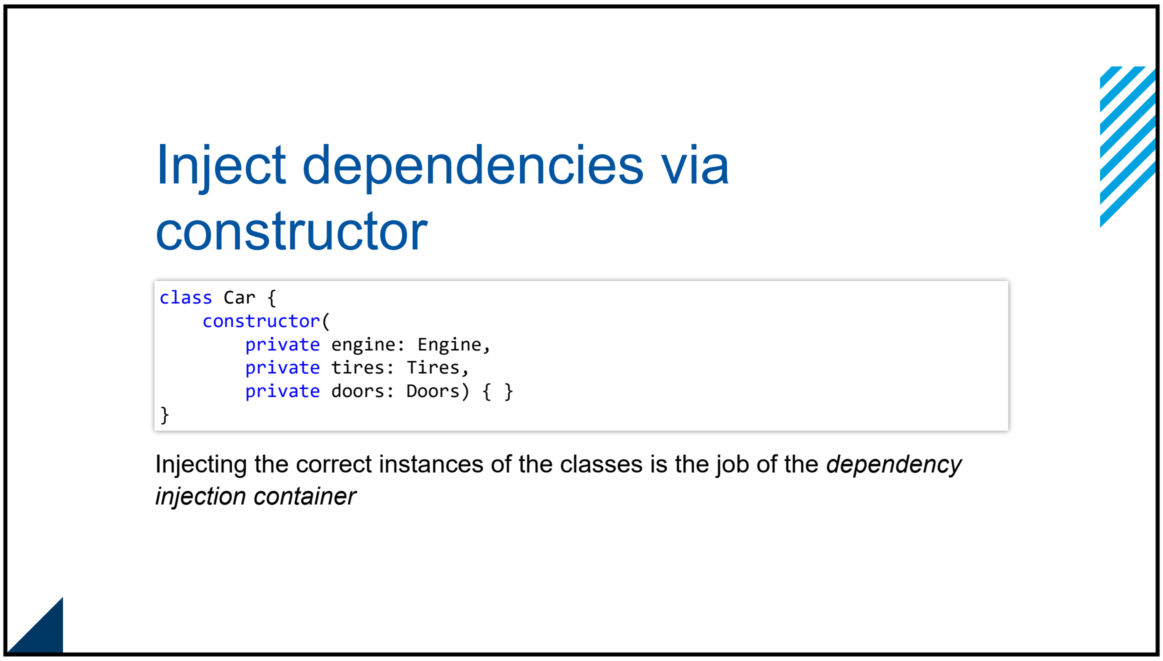
- It's a form of inversion of control
- It's about **expressing a need**
- You tell Angular your component needs to perform AJAX requests and Angular supplies you with something that can do just that



# What DI solves

```
class Car {
    constructor() {
        this.engine = new Engine();
        this.tires = Tires.getInstance();
        this.doors = app.get('doors');
    }
}
```

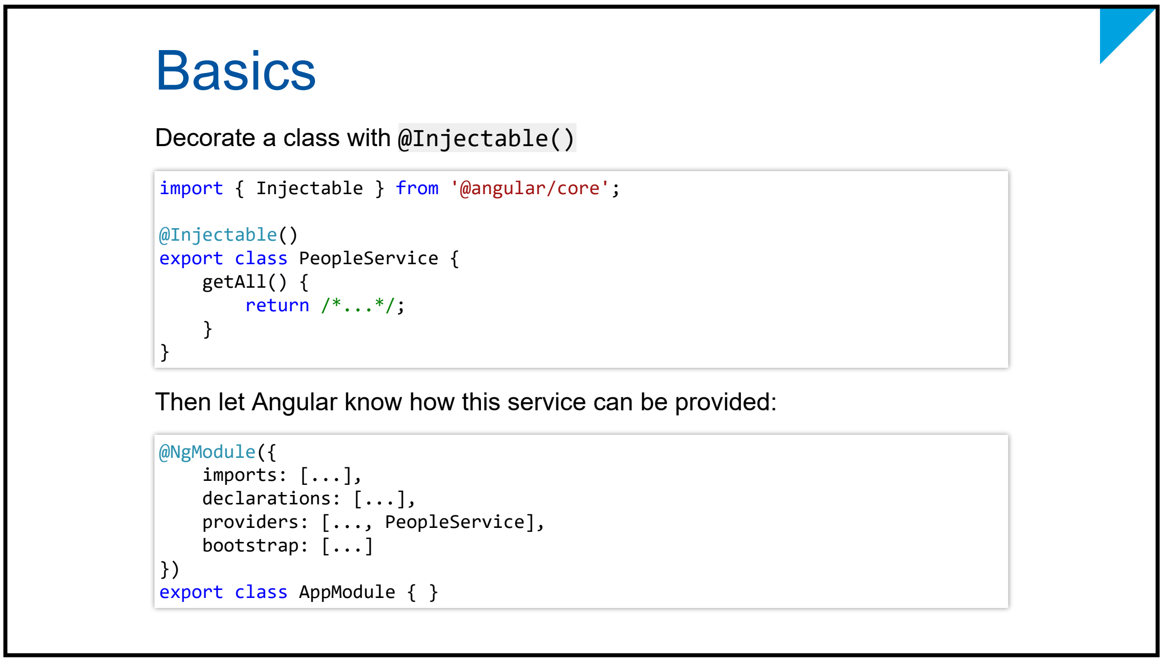
- Doing this everywhere in your application will lead to a lot of rework
- Mocking away dependencies becomes horribly complicated



# Inject dependencies via constructor

```
class Car {
    constructor(
        private engine: Engine,
        private tires: Tires,
        private doors: Doors) { }
}
```

Injecting the correct instances of the classes is the job of the *dependency injection container*



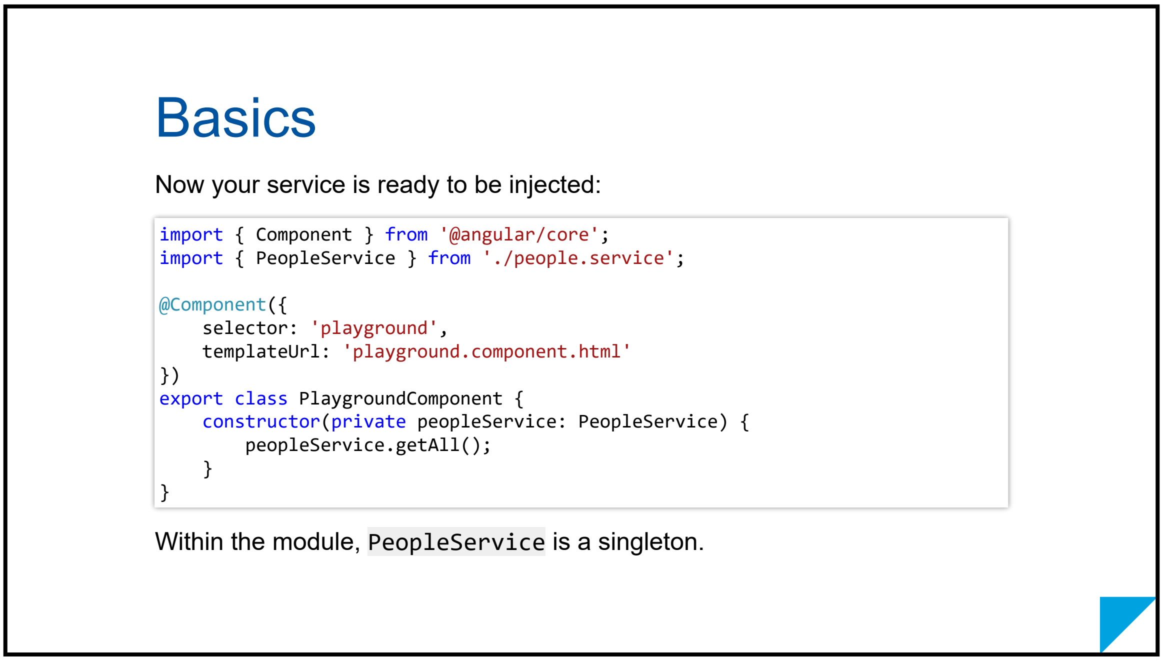
# Basics

Decorate a class with `@Injectable()`

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

@Injectable()
export class PeopleService {
    getAll() {
        return /*...*/;
    }
}
```

Then let Angular know how this service can be provided:

```
@NgModule({
    imports: [...],
    declarations: [...],
    providers: [..., PeopleService],
    bootstrap: [...]
})
export class AppModule { }
```

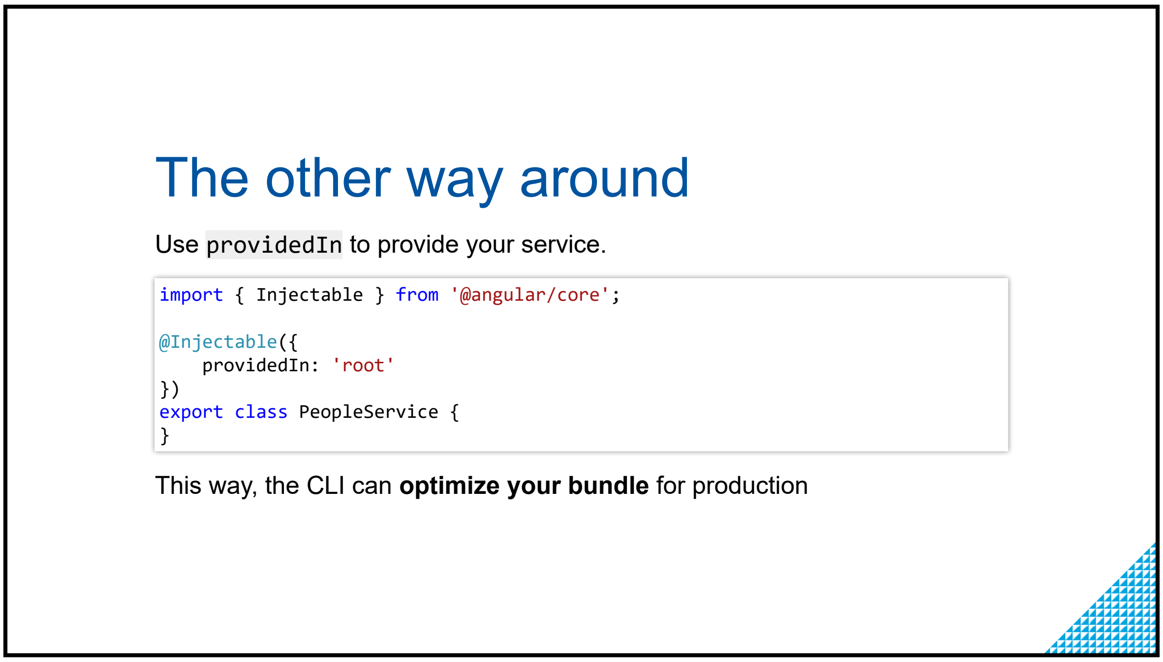


# Basics

Now your service is ready to be injected:

```
import { Component } from '@angular/core';
import { PeopleService } from './people.service';

@Component({
    selector: 'playground',
    templateUrl: 'playground.component.html'
})
export class PlaygroundComponent {
    constructor(private peopleService: PeopleService) {
        peopleService.getAll();
    }
}
```

Within the module, `PeopleService` is a singleton.

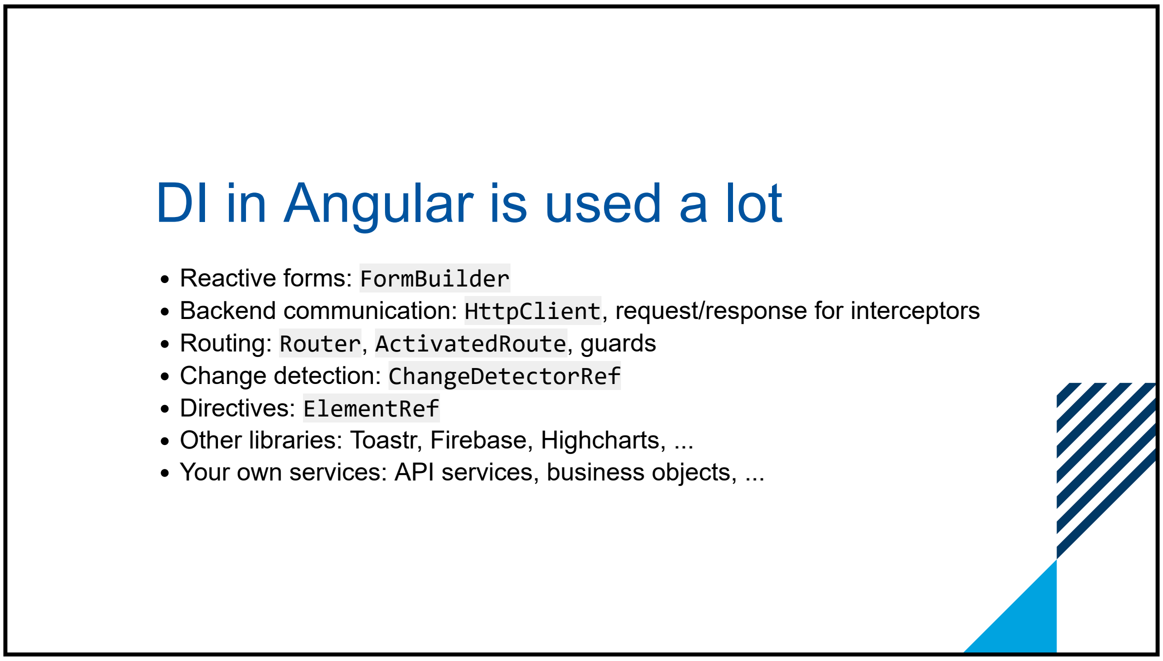


## The other way around

Use `providedIn` to provide your service.

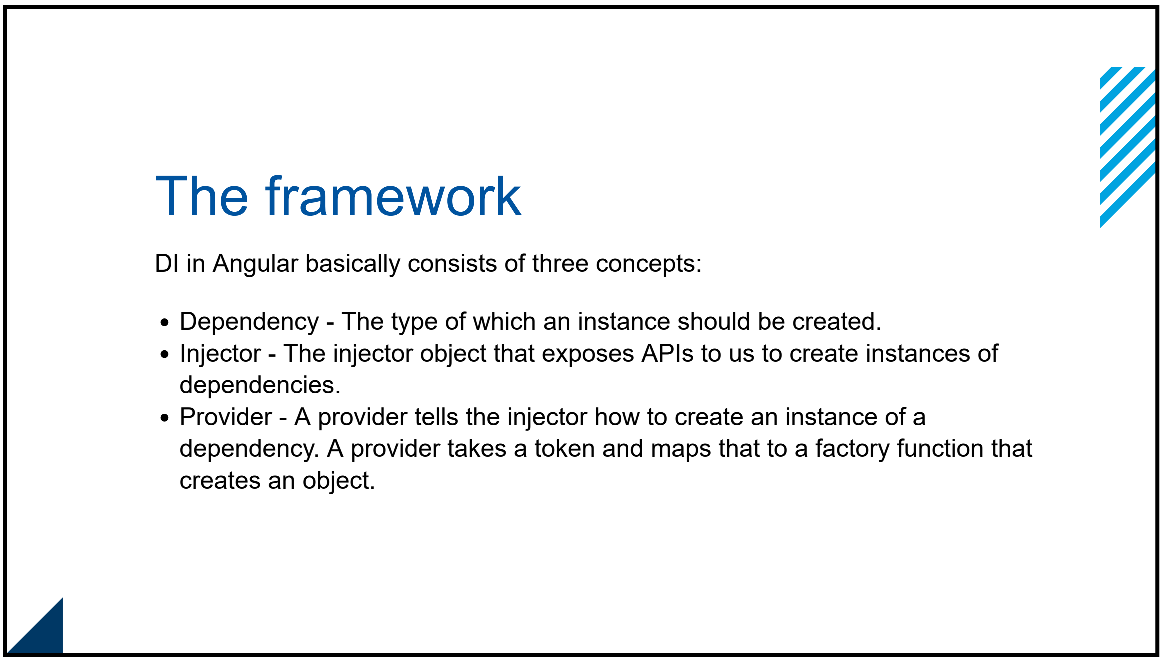
```
import { Injectable } from '@angular/core';

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

This way, the CLI can **optimize your bundle** for production



## DI in Angular is used a lot

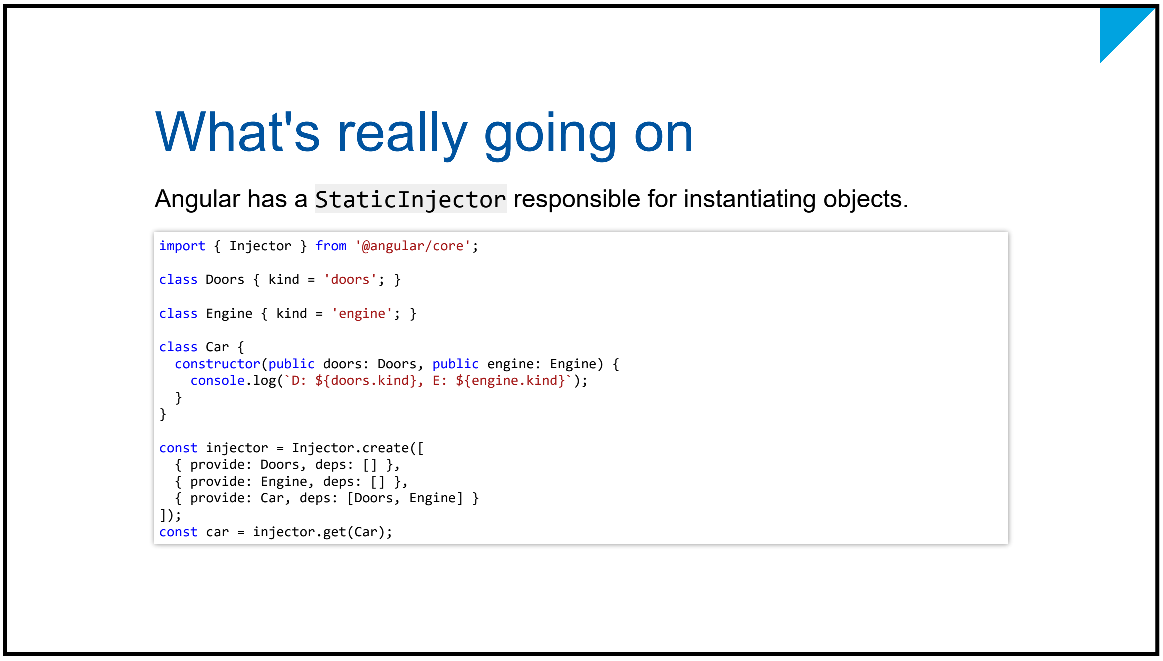
- Reactive forms: `FormBuilder`
- Backend communication: `HttpClient`, request/response for interceptors
- Routing: `Router`, `ActivatedRoute`, guards
- Change detection: `ChangeDetectorRef`
- Directives: `ElementRef`
- Other libraries: Toastr, Firebase, Highcharts, ...
- Your own services: API services, business objects, ...



## The framework

DI in Angular basically consists of three concepts:

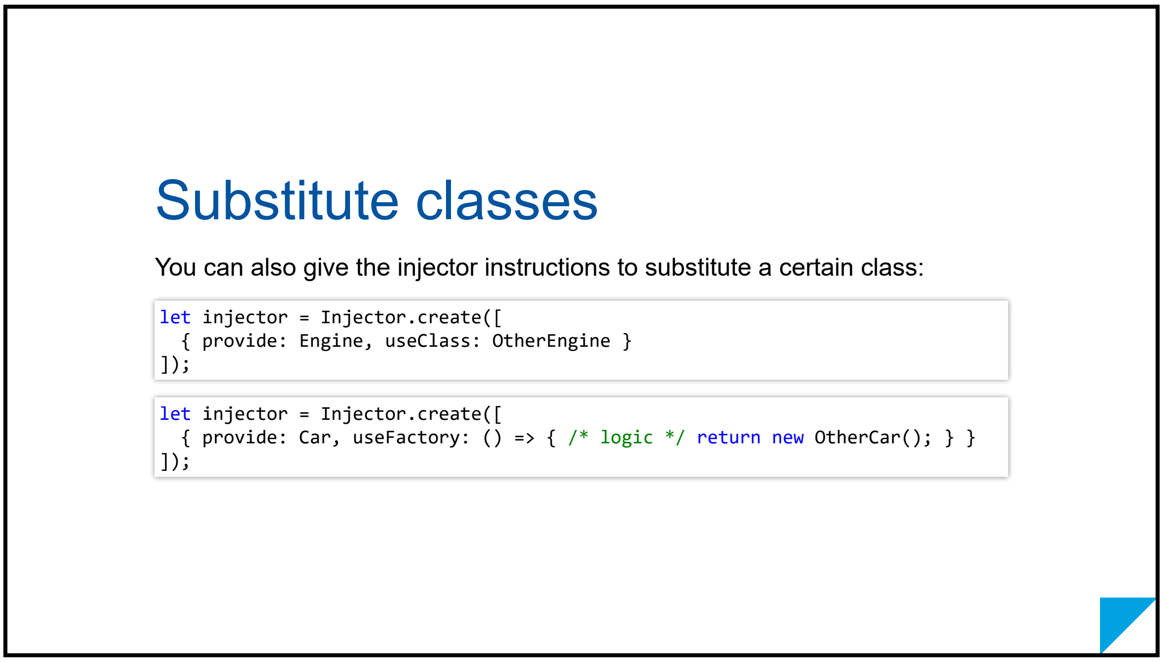
- Dependency - The type of which an instance should be created.
- Injector - The injector object that exposes APIs to us to create instances of dependencies.
- Provider - A provider tells the injector how to create an instance of a dependency. A provider takes a token and maps that to a factory function that creates an object.



## What's really going on

Angular has a `StaticInjector` responsible for instantiating objects.

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

class Doors { kind = 'doors'; }

class Engine { kind = 'engine'; }

class Car {
  constructor(public doors: Doors, public engine: Engine) {
    console.log(`D: ${doors.kind}, E: ${engine.kind}`);
  }
}

const injector = Injector.create([
  { provide: Doors, deps: [] },
  { provide: Engine, deps: [] },
  { provide: Car, deps: [Doors, Engine] }
]);
const car = injector.get(Car);
```

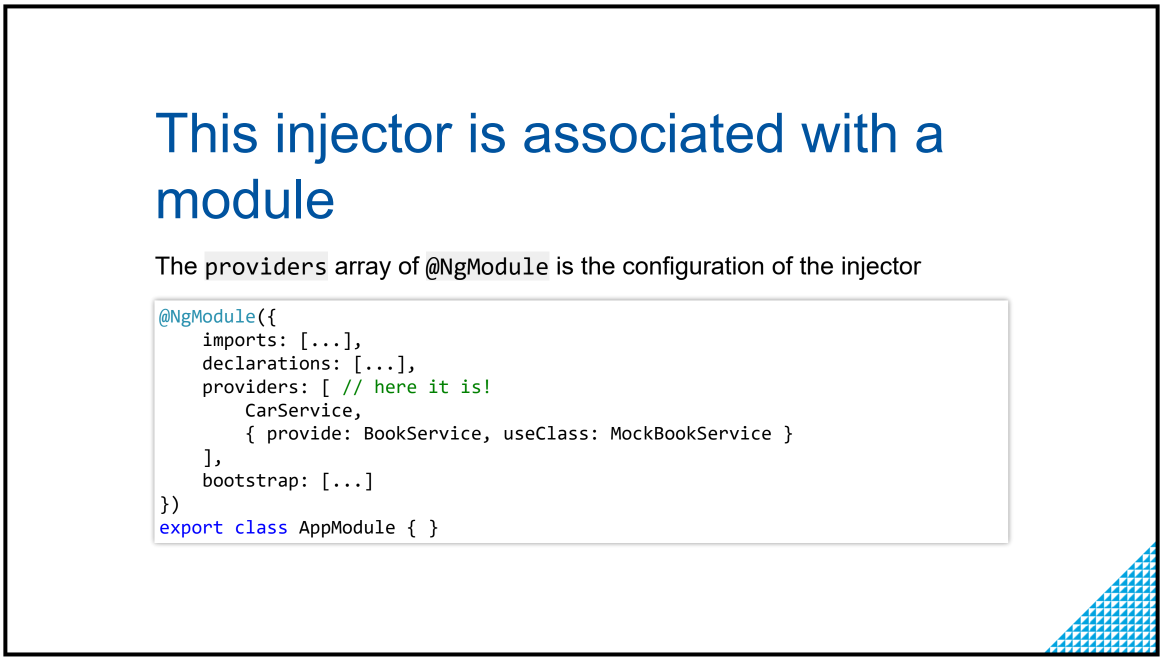


## Substitute classes

You can also give the injector instructions to substitute a certain class:

```
let injector = Injector.create([
  { provide: Engine, useClass: OtherEngine }
]);
```

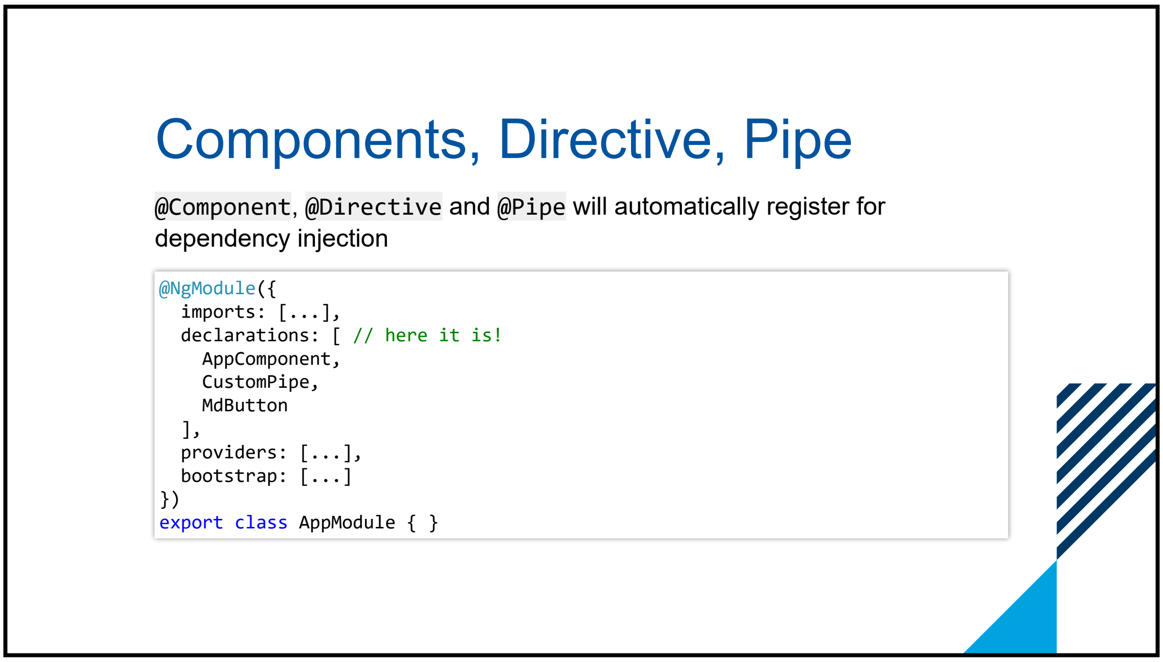
```
let injector = Injector.create([
  { provide: Car, useFactory: () => { /* logic */ return new OtherCar(); } }
]);
```



## This injector is associated with a module

The `providers` array of `@NgModule` is the configuration of the injector

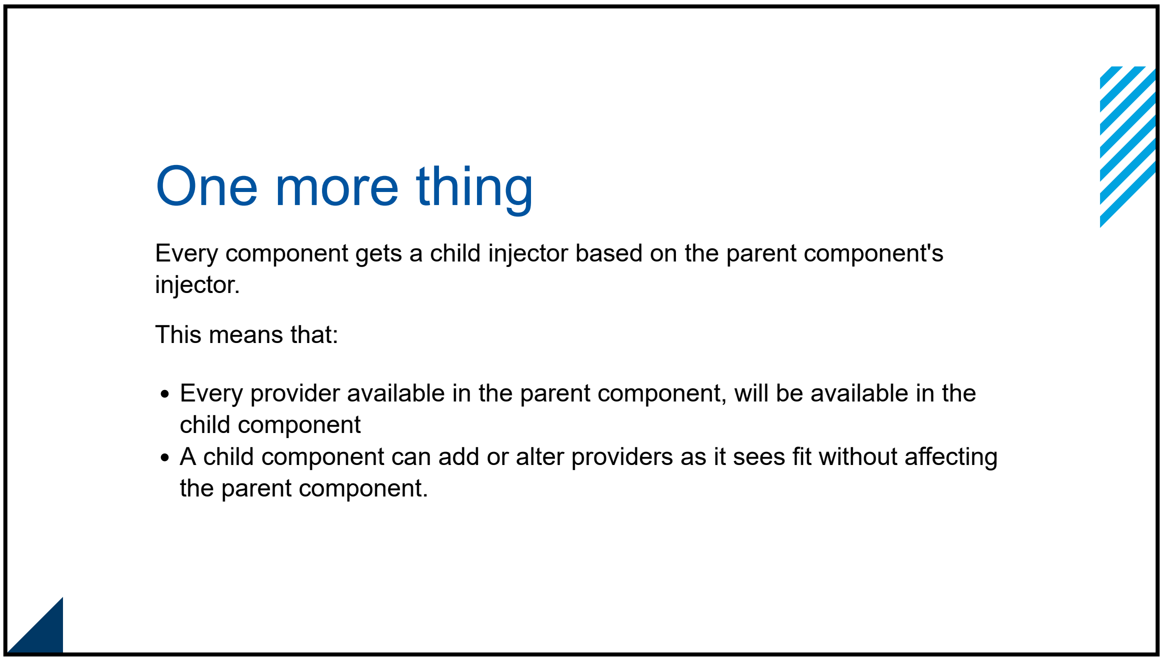
```
@NgModule({
    imports: [...],
    declarations: [...],
    providers: [ // here it is!
        CarService,
        { provide: BookService, useClass: MockBookService }
    ],
    bootstrap: [...]
})
export class AppModule { }
```



# Components, Directive, Pipe

`@Component`, `@Directive` and `@Pipe` will automatically register for dependency injection

```
@NgModule({
  imports: [...],
  declarations: [ // here it is!
    AppComponent,
    CustomPipe,
    MdButton
  ],
  providers: [...],
  bootstrap: [...]
})
export class AppModule { }
```

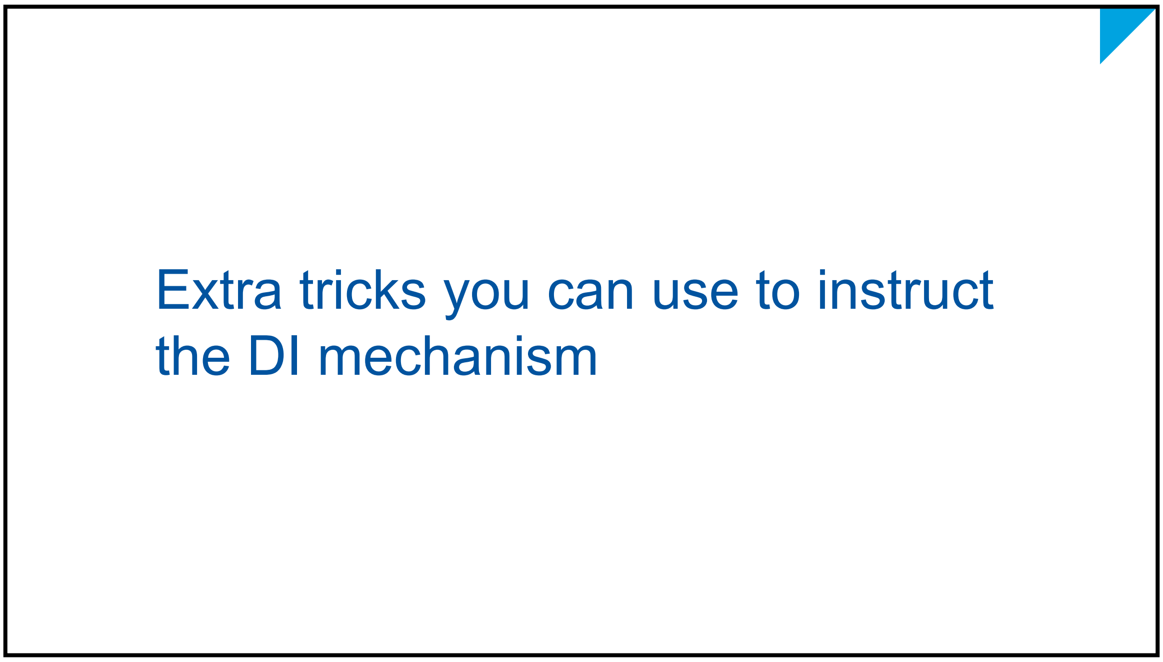


# One more thing

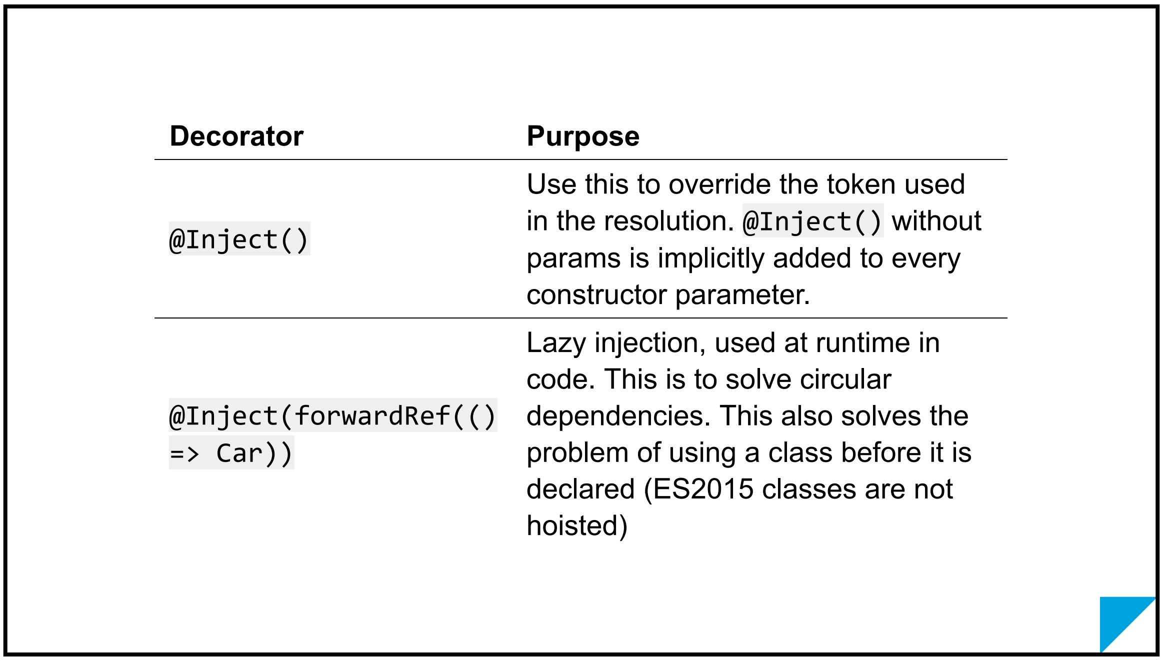
Every component gets a child injector based on the parent component's injector.

This means that:

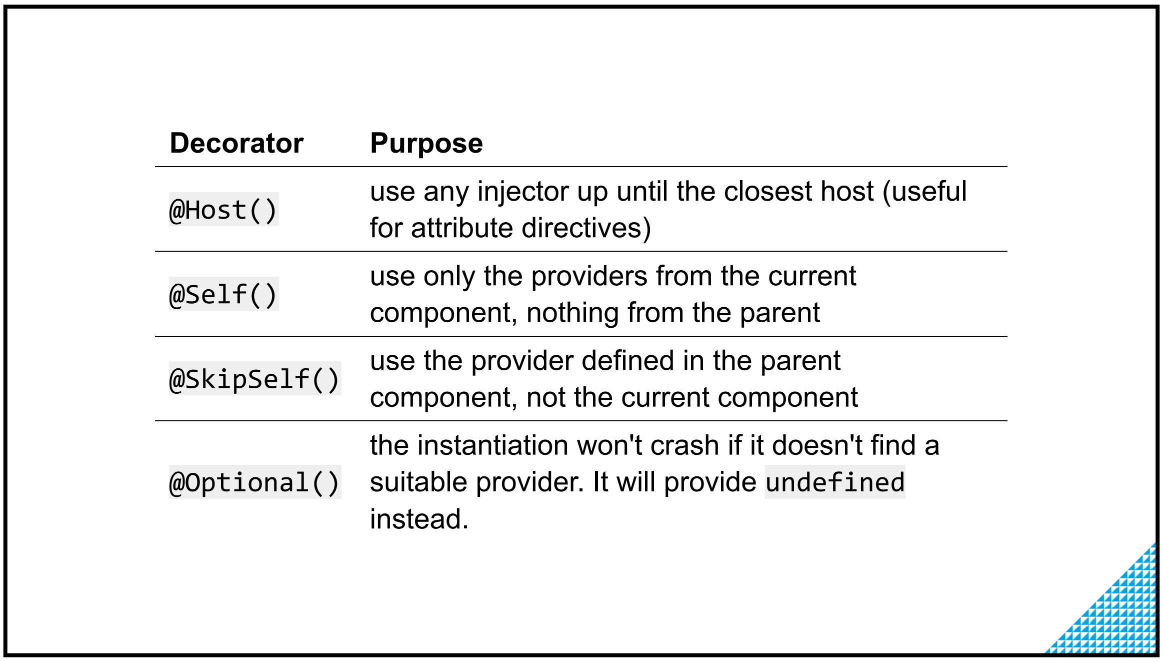
- Every provider available in the parent component, will be available in the child component
- A child component can add or alter providers as it sees fit without affecting the parent component.



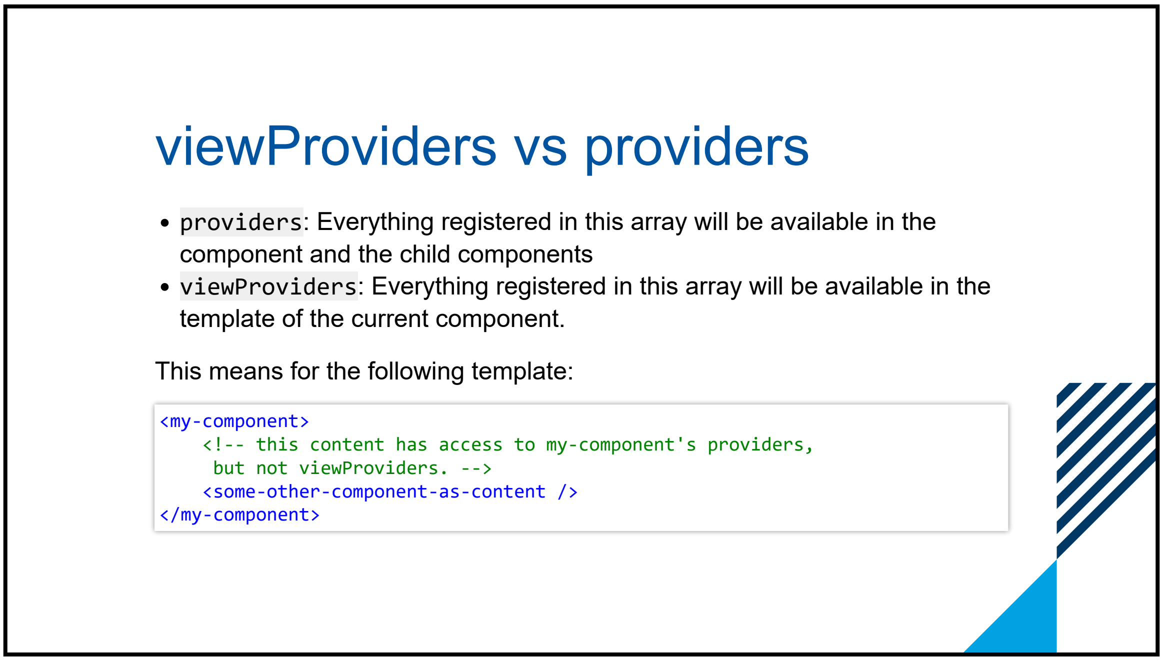
## Extra tricks you can use to instruct the DI mechanism



| Decorator | Purpose |
|---|---|
| `@Inject()` | Use this to override the token used in the resolution. `@Inject()` without params is implicitly added to every constructor parameter. |
| `@Inject(forwardRef(() => Car))` | Lazy injection, used at runtime in code. This is to solve circular dependencies. This also solves the problem of using a class before it is declared (ES2015 classes are not hoisted) |



| Decorator | Purpose |
|---|---|
| `@Host()` | use any injector up until the closest host (useful for attribute directives) |
| `@Self()` | use only the providers from the current component, nothing from the parent |
| `@SkipSelf()` | use the provider defined in the parent component, not the current component |
| `@Optional()` | the instantiation won't crash if it doesn't find a suitable provider. It will provide `undefined` instead. |

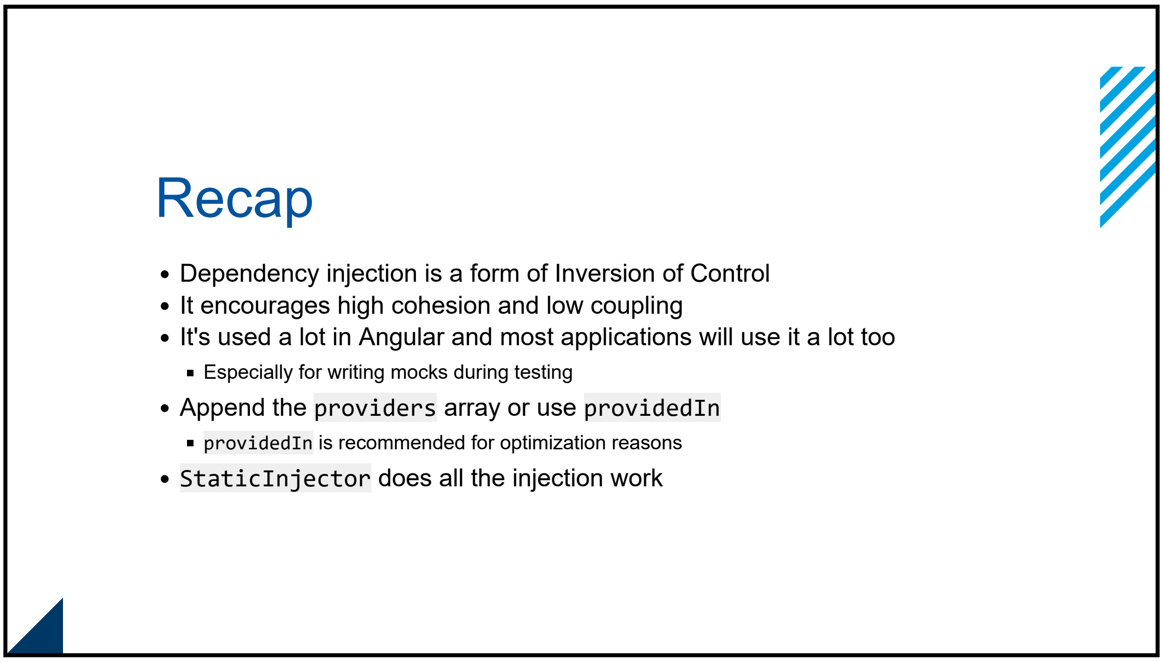


# viewProviders vs providers

- `providers`: Everything registered in this array will be available in the component and the child components
- `viewProviders`: Everything registered in this array will be available in the template of the current component.

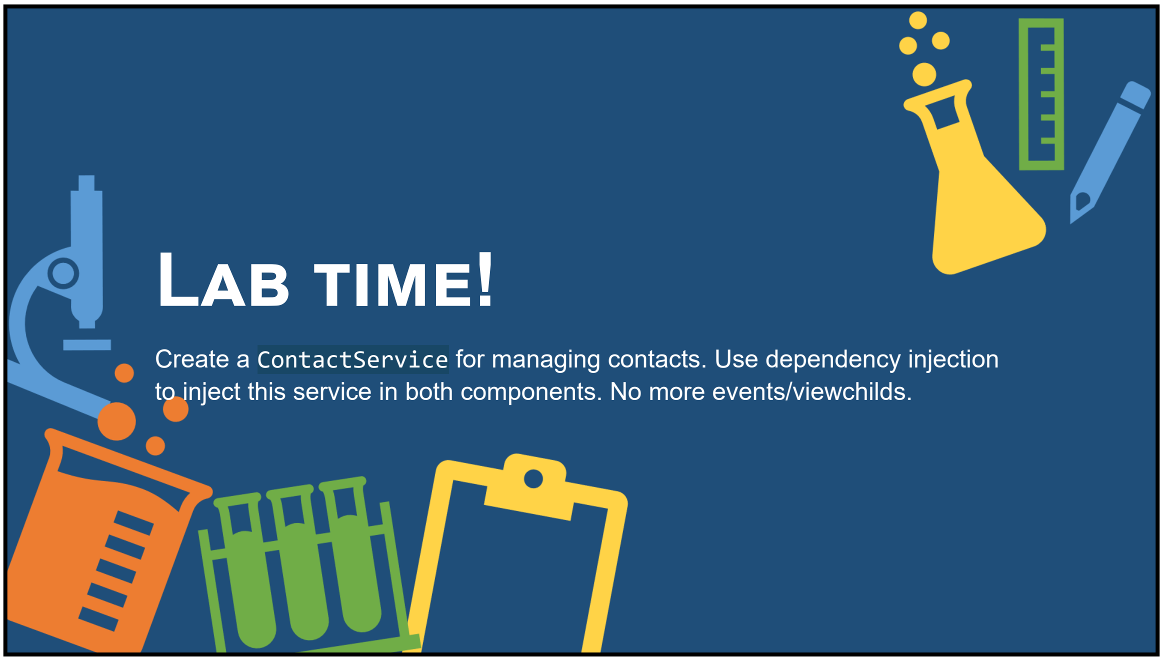
This means for the following template:

```
<my-component>
    <!-- this content has access to my-component's providers,
     but not viewProviders. -->
    <some-other-component-as-content />
</my-component>
```



# Recap

- Dependency injection is a form of Inversion of Control
- It encourages high cohesion and low coupling
- It's used a lot in Angular and most applications will use it a lot too
  - Especially for writing mocks during testing
- Append the `providers` array or use `providedIn`
  - `providedIn` is recommended for optimization reasons
- `StaticInjector` does all the injection work



# Lab time!

Create a `ContactService` for managing contacts. Use dependency injection to inject this service in both components. No more events/viewchilds.